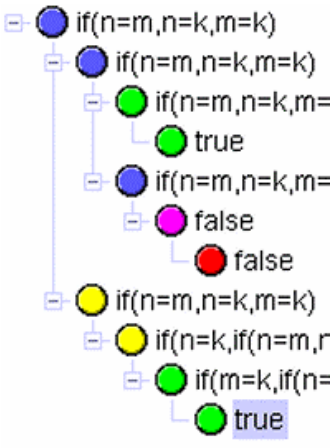



# VeriFun
# User Guide

Technical Report VFR 02/01

( Last Update October 27, 2002 )

*Christoph Walther*
*Stephan Schweitzer*

Programmiermethodik
Fachbereich Informatik

TECHNISCHE
UNIVERSITÄT
DARMSTADT

# About VeriFun

This document describes how to operate the VeriFun system. Formal background is provided only to the extend necessary for a proper use of the system. A system user should be familiar with the basic concepts and notions of mathematical logic, formal methods and theorem proving.

This document is organized analogue to the menu-structure of VeriFun to ease the navigation when searching for help. It is not a tutorial or a scientific report. Instead it is designed as an online reference manual (with hyperlinks indicated in blue) and therefore contains repetitions but no explanations other than required to operate the system and gain maximal use of it.

Novices to VeriFun are recommended to read

- *About VeriFun*,
- *About File*,
- *About Program*,
- *About Proof*,
- *About Evaluation*, and
- *About Window*

and also *The* VeriFun *Tutorial* before beginning to work with the system. The system, installation information and *The* VeriFun *Tutorial* can be obtained from

*http://www.informatik.tu-darmstadt.de/pm/verifun/* .

Since this document underlies a permanent revision, this web-page should be also visited to see whether a more recent version exists. Questions, comments and critique may be directed to

*verifun@informatik.tu-darmstadt.de* .

This document does not apply to versions older than VeriFun 2.5.5.

## *V*eriFun *at a Glance*

✓eriFun is a semi-automated system for the verification of statements about programs written in a functional programming language.

In a typical session with the system, a user

- defines a *program* by stipulating the *data structures* and the *procedures* of the program using ✓eriFun's *language editor*,
- defines *statements* about the data structures and procedures of the program using ✓eriFun's *language editor*,
- verifies these statements and the termination of the procedures using ✓eriFun's *proof editor*.

The system also provides the usual *file*-commands to save and reload intermediate work if required.

✓eriFun consists of several *fully-automated* routines for *theorem proving* and for the *formation of hypotheses* to support verification. However, as most problems to be solved in program verification are not semi-decidable, these routines *must* fail for certain verification tasks. In such a case, the user may step in to guide the system for the continuation of a proof.

✓eriFun is designed as an interactive system, where, however, the automated routines substitute the human expert in striving for a proof until they fail. This architecture cares for a user friendly interface, as he or she is relieved from deciding whether and when to switch between an automated and an interactive operation mode.

## *Functional Programs*

Functional programs are formulated in a programming language called ✓afp (abbreviating ✓eriFun's *annotated functional programs*). ✓afp is a „simple" functional programming language, as no higher-order language constructs like *procedures-as-parameters*, *mapping functions* etc. are provided.

An ✓afp – expression either defines
- a *data structure*,
- a (functional) *procedure*, or
- a *lemma*.

A set of data structures and procedures form a *functional program*. The lemmas are used to formulate statements about the data structures and procedures, yielding an *annotated* functional program. In this document,
- „program" refers to a ✓afp – program being annotated or not,
- „lemma" refers to a statement being proved, refuted or unproved.

The program which results from the user definitions in a ✓eriFun-session is called the *actual program*. The *actual program* is the program which is displayed in the *Program Window*.

### *Syntax*

√afp – expressions are based on *many-sorted first-order terms* over a *signature* for function- and sort symbols. This signature is extended incrementally as new definitions enter the system.

The signature at any time includes

- the sort symbol `bool`,
- the function symbols `false` : → `bool` and `true` : → `bool`,
- the sort symbol `nat`,
- the function symbols `0` : → `nat`, `succ` : `nat` → `nat` and `pred` : `nat` → `nat`,
- the infix function symbol `>` : `nat` × `nat` → `bool`,
- the function symbols $\texttt{if}_{\texttt{sort}}$ : `bool` × `sort` × `sort` → `sort` for each sort symbol `sort` of the signature (which are written without subscripts),
- the infix function symbols $=_{\texttt{sort}}$ : `sort` × `sort` → `bool` for each sort symbol `sort`≠`bool` of the signature (which are written without subscripts).

Each √afp – expression consists of

- a *keyword*,
- a *name*, and
- a *body*,

where a *name* is an identifier beginning with a letter from {A,…,Z,a,…,z} which may be followed by a sequence of letters, digits from {0,…,9} and symbols from {-, _, ., #, !, &, *, +, ?, ;}.

***Data structures***

are defined in a *constructor-selector*-style discipline by expressions of the form

```
structure SORT <= BODY ,
```

where

- `structure` is a *keyword*,
- `SORT` is an identifier not used so far defining the *name* of the data structure,
- `BODY` is a non-empty sequence of *structure patterns* defining the *body* of the data structure,
- a *structure pattern* is an expression of the form $\texttt{cons(sel}_1\texttt{: sort}_1\texttt{,...,sel}_n\texttt{: sort}_n\texttt{)}$ and introduces one *constructor* function symbol $\texttt{cons : sort}_1 \times ... \times \texttt{sort}_n \to \texttt{SORT}$ and $n \geq 0$ *selector* function symbols $\texttt{sel}_i\texttt{: SORT} \to \texttt{sort}_i$ not used so far,
- the identifiers $\texttt{sort}_i$ either are sort symbols from the signature, i.e. names of some data structures already defined, or equal `SORT`.

A data structure definition extends the signature by the new sort symbol `SORT` and the new function symbols `cons`, $\texttt{sel}_1$, ... , $\texttt{sel}_n$. A (constructor) function symbol $\texttt{cons : sort}_1 \times ... \times \texttt{sort}_n \to \texttt{SORT}$ is *reflexive* iff $\texttt{SORT=sort}_i$ for some *i*, and is *irreflexive* otherwise. It is demanded that at least *one irreflexive* constructor exists for `SORT`.

`SORT` is *recursively* defined if at least *one reflexive* constructor exists for `SORT`. Each data structure `SORT` is assigned a so-called *witness-term* $\nabla\texttt{SORT}$, which is defined as the application of the leftmost irreflexive constructor in the definition of `SORT` applied to the witness-terms of its argument sorts.

For instance,

```
structure bool <= false,true
```

is predefined in ✓`eriFun` and defines a non-recursive data structure with witness-term $\nabla\texttt{bool} = \texttt{false}$.

```
structure nat <= 0,succ(pred:nat)
```

is also predefined in ✓eriFun and defines a recursive data structure with witness-term ∇nat = 0.

```
structure sexpr <=
atom(index:nat),nil,cons(car:sexpr,cdr:sexpr)
```

defines a recursive data structure with ∇sexpr = atom(0).

***Procedures***

are defined by expressions of the form

$$\texttt{function PROC(}x_1\texttt{:}sort_1\texttt{,...,}x_n\texttt{:}sort_n\texttt{):sort <= BODY ,}$$

where

- function is a *keyword*,
- PROC is an identifier not used so far defining the *name* of the procedure,
- $n \geq 1$, $sort_1$, ... ,$sort_n$ are sort symbols from the signature and the $x_i$ are distinct variable symbols of the sorts $sort_i$ respectively, called the *formal parameters* of the procedure PROC,
- BODY either is the symbol ? or is a first-order term of sort sort over the signature expanded by PROC, which contains no other variable symbols than the formal parameters, defining the *body* of the procedure.

A procedure definition extends the signature by the function symbol PROC. PROC is called an *undefined* function symbol and denotes an *undefined procedure* iff BODY=?. Otherwise PROC is *recursively* defined, provided BODY contains one procedure call PROC(...) at least, and is *non-recursively defined* else.

When defining procedures

- *shorthand notation* like x,y:sexpr for x:sexpr,y:sexpr,
- *programming notation* if ... then ... else ... fi instead of *functional notation* if( ... , ... , ... ), and
- *numbers* ≤ *42*, e.g. 2 instead of succ(succ(0)),

may be used.

For instance,

```
function is.five(n:nat):bool <= n=5
```

defines a non-recursive procedure.

```
function member(x,y:sexpr):sexpr <=
  if y=cons(car(y),cdr(y))
    then if x=car(y)
           then y
           else member(x,cdr(y))
         fi
    else nil
  fi
```

defines a recursive procedure, and

```
function minus(x,y:nat):nat <=
  if x=0
    then 0
    else if y=0
           then x
           else minus(pred(x),pred(y))
         fi
  fi
```

defines a recursive procedure too.

An expression like

```
function result(x:nat):bool <= ?
```

defines an undefined procedure and `result` is an undefined function symbol.

***Quantifications***

adopt the rôle of formulas in ✓`eriFun` and are represented by *boolean terms*. *Connectives* are expressed by *conditionals* using

| | | |
|---|---|---|
| `if(a,false,true)` | for | $\neg$`a` , |
| `if(a,true,b)` | for | `a` $\vee$ `b` , |
| `if(a,false,b)` | for | $\neg$`a` $\wedge$ `b`, |
| `if(a,b,true)` | for | `a` $\rightarrow$ `b` , |
| `if(a,b,false)` | for | `a` $\wedge$ `b` , and |
| `if(a,b,if(b,false,true))` | for | `a` $\leftrightarrow$ `b` . |

A boolean term `b` is called

- an *atom* or a *positive literal* iff `b` does not contain an `if`-symbol,
- a *negative literal* iff `b` = `if(b',false,true)`, where `b'` is an atom,
- a *literal* iff `b` is a positive or negative literal,
- an *equation* iff `b` = `left=right` and `b` is an atom.

A finite set of literals is called a *clause*. Boolean terms may be *universally quantified* yielding so-called *quantifications* of the form

$$\texttt{all}\ \texttt{x}_1\texttt{:sort}_1\texttt{,}\ldots\texttt{,x}_n\texttt{:sort}_n\ \texttt{TERM}\ ,$$

where

- $n \geq 0$,
- $\texttt{sort}_1, \ldots, \texttt{sort}_n$ are sort symbols from the signature,
- the $\texttt{x}_i$ are distinct variable symbols of the sorts $\texttt{sort}_i$, and
- `TERM` is a first-order term of sort `bool` over the signature.

A quantification is *closed* iff `TERM` contains no other variable symbols than the variables $\texttt{x}_i$. Closed quantification are used to define

- lemmas,
- termination hypotheses, and
- recursion elimination formulas.

*Non-closed* quantification are used to represent induction hypotheses.

***Lemmas***
are defined by expressions of the form

```
lemma LEM <= BODY ,
```

where
- `lemma` is a *keyword,*
- `LEM` is an identifier not used so far defining the lemma's *name,* and
- `BODY` is a closed quantification defining the *body* of the lemma.

When defining lemmas
- *shorthand notation* like `x,y:sexpr` for `x:sexpr,y:sexpr`,
- *programming notation* `if … then … else … fi` instead of *functional notation* `if( … , … , … )`, and
- *numbers* ≤ *42,* e.g. `2` instead of `succ(succ(0))`,

may be used.

For instance,

```
lemma member_transitivity <= all x,y,z:sexpr
   if(member(x,y)=nil,
      true,
      if(member(y,z)=nil,
         true,
         if(member(x,z)=nil,false,true)))
```

defines a lemma about member.

## *Semantics*

The purpose of ✓`eriFun` is

- to verify lemmas, i.e. statements about the data structures and procedures in a ✓`afp` – program, and
- to prove the termination of the procedures in a ✓`afp` – program

both wrt. the semantics defined for ✓`afp`.

### *Data structures and Procedures*

An *operational semantics* for ✓`afp` is defined by an *interpreter*

$$\mathit{eval} : \mathcal{T} \mapsto \mathcal{V}al \cup \mathcal{S}tuck$$

which maps the members of the set $\mathcal{T}$ of variable-free terms over the signature of the actual program to a set $\mathcal{V}al$ of *values* and a set $\mathcal{S}tuck$ of so-called *stucked evaluations*. The set $\mathcal{V}al$ consists of terms build with *constructor* function symbols only. The set $\mathcal{S}tuck$ consists of variable-free terms over the signature of the actual program which contain one *undefined* function symbol at least.

The *value* $\mathit{eval}$(`t`) of a term `t` is obtained by the usual interpretation of *conditionals* `if(...)`, *equations* `...=...` , selectors as the *inverse functions* of constructors and *procedure calls* using the *call-by-value* parameter passing discipline.

Formally, the interpreter *eval* is defined as a *partial* mapping satisfying

- *eval*(q)=q, if q$\in \mathcal{V}al$,
- *eval*(if(cond,then,else))=*eval*(then), if *eval*(cond)=true,
- *eval*(if(cond,then,else))=*eval*(else), if *eval*(cond)=false,
- *eval*(left=right)=true, if *eval*(left)$\in \mathcal{V}al \cup \mathcal{S}tuck$, *eval*(right) $\in \mathcal{V}al \cup \mathcal{S}tuck$, and *eval*(left)=*eval*(right),
- *eval*(left=right)=false, if *eval*(left)$\in \mathcal{V}al$, *eval*(right)$\in \mathcal{V}al$, and *eval*(left)$\neq$*eval*(right),
- *eval*($\mathtt{sel}_i(\mathtt{cons}(\mathtt{t}_1,\ldots,\mathtt{t}_n))$)=*eval*($\mathtt{t}_i$), if *eval*($\mathtt{t}_1$)$\in \mathcal{V}al \cup \mathcal{S}tuck$, … , *eval*($\mathtt{t}_n$)$\in \mathcal{V}al \cup \mathcal{S}tuck$ and cons(…, $\mathtt{sel}_i$ : $\mathtt{sort}_i$, …) is a structure pattern,
- *eval*($\mathtt{sel}_i(\mathtt{cons}(\mathtt{t}_1,\ldots,\mathtt{t}_n))$)=$\nabla\mathtt{sort}_i$, if *eval*($\mathtt{t}_1$)$\in \mathcal{V}al \cup \mathcal{S}tuck$, … , *eval*($\mathtt{t}_n$)$\in \mathcal{V}al \cup \mathcal{S}tuck$, cons‘(…, $\mathtt{sel}_i$ : $\mathtt{sort}_i$, …) is a structure pattern and cons$\neq$cons‘,
- *eval*(f($\mathtt{t}_1$,…,$\mathtt{t}_n$))=f(*eval*($\mathtt{t}_1$),…,*eval*($\mathtt{t}_n$)), if f is a *constructor*, a *selector* or an *undefined* function symbol,
- *eval*(f($\mathtt{t}_1$,…,$\mathtt{t}_n$))=*eval*(BODY‘), if f is defined by function f($\mathtt{x}_1$: $\mathtt{sort}_1$,…,$\mathtt{x}_n$:$\mathtt{sort}_n$):sort <= BODY, BODY$\neq$? and BODY‘ is obtained from BODY by replacing the formal parameters $\mathtt{x}_i$ in BODY with the *evaluated* actual parameters *eval*($\mathtt{t}_i$).

*eval* is a *partial* mapping, because procedures need not to *terminate* and then *eval*(t) may be *undefined* if t contains a call of such a procedure. However, this is not to confuse with *eval*(t) containing *undefined function symbols*.

***Quantifications and Lemmas***

A closed quantification `all` $\texttt{x}_1\texttt{:sort}_1\texttt{,}\ldots\texttt{,x}_n\texttt{:sort}_n$ `TERM` is *true* iff the value of the term `TERM'`, obtained from `TERM` by substituting values for the universally quantified variables $\texttt{x}_i$, is `true` no matter which values are substituted for the variables.

The semantics of closed quantifications respect the interpretation of the data structures and procedures and is formally defined by:

A closed quantification

$$\texttt{all}\ \texttt{x}_1\texttt{:sort}_1\texttt{,}\ldots\texttt{,x}_n\texttt{:sort}_n\ \texttt{TERM}\ \text{ is } \textit{true}$$

$$\leftrightarrow$$

$$\textit{eval}(\texttt{TERM[x}_1\texttt{/q}_1\texttt{,}\ldots\texttt{,x}_n\texttt{/q}_n\texttt{]}) = \texttt{true for all q}_1\texttt{,}\ldots\texttt{,q}_n \in \mathcal{V}al,$$

where $\texttt{TERM[x}_1\texttt{/q}_1\texttt{,}\ldots\texttt{,x}_n\texttt{/q}_n\texttt{]}$ is obtained from `TERM` by replacing the variables $\texttt{x}_i$ in `TERM` with the values $\texttt{q}_i$ of sort $\texttt{sort}_i$.

A lemma `lemma LEM <= BODY` is *true* iff `BODY` is *true*.

The purpose of ✓`eriFun` is to verify the truth of a lemma by computing a proof for it.

## *Termination of Procedures*

Using ✓eriFun, the *total* correctness of procedures can be verified, i.e. only those lemmas can be verified, the bodies of which only contain calls of procedures whose *termination* has been established.

A procedure of the actual program

$$\texttt{function PROC(}x_1\texttt{:}sort_1\texttt{,...,}x_n\texttt{:}sort_n\texttt{):}sort\texttt{ <= BODY ,}$$

*terminates* iff $eval(\texttt{PROC(}q_1\texttt{,...,}q_n\texttt{)}) \in \mathcal{V}al \cup \mathcal{S}tuck$ for all values $q_1,...,q_n$ of the actual program.

Consequently, each undefined procedure terminates, and a non-recursively defined procedure terminates iff the procedure body contains only calls of terminating procedures.

A recursively defined procedure *terminates* iff (*i*) only terminating procedures are non-recursively called in the procedure body and (*ii*) some *well-founded relation* exists such that for *each* recursive call, the arguments of the *initial call* are greater (wrt. the well-founded relation) than the arguments of the *recursive call* whenever the conditions leading to that recursive call are satisfied.

To verify the termination of a procedure

*(i)* *termination hypotheses*, i.e. closed quantifications whose truth entail that the procedure terminates, must be *formulated*, and then

*(ii)* the *termination hypotheses* must be *proved*.

Problem *(ii)* is solved in the same way as lemmas are proved in ✓eriFun. Problem *(i)* is solved by ✓eriFun's automated termination analysis or by the user, if the systems fails.

***User guided Termination Proofs***

In ✓eriFun, all user guided termination proofs are based on the *well-founded* >-relation on natural numbers, represented by the predefined procedure >.

**Measure Terms**

To prove the termination of a recursively defined procedure given as

$$\texttt{function PROC(x}_1\texttt{:sort}_1\texttt{,...,x}_n\texttt{:sort}_n\texttt{):sort <= BODY ,}$$

a so-called *measure term* M must be provided for PROC, where M is a nat-term containing no other variables than the formal parameters of the procedure. A measure term represents what is commonly known as a *termination function*. The set of variables in M is called a *recursion position set* for PROC.

Based on a given measure term, the system generates a *termination hypothesis* for each *recursive call* in BODY, formulated with procedure calls of >. Each termination hypothesis expresses that the termination requirement is satisfied for that recursive call. The termination of PROC is verified, if each termination hypotheses generated for PROC is proved.

Instead of providing *one* measure term, a *sequence of measure terms* can also be stipulated, and then termination hypotheses based on the *lexicographic ordering* imposed by > are generated. In such a case, the set of all variables in the sequence yield a *recursion position set* for the procedure.

Also *recursively* defined data structures are assigned a recursion position set consisting of one variable symbol only, and the *empty set* is a the recursion position set for all *non-recursively* defined data structures and procedures.

**Optimal Sets of Recursion Position Sets**

It may happen that *different independent termination proofs* for a procedure exist. While it does not matter wrt. total correctness how termination is justified, the information obtained from different termination proofs is crucial for the system's theorem proving capabilities, as *induction axioms* are automatically derived from them (and from the definition of the procedure). In order to gain the maximal use of the termination analysis, a set of measure terms should be provided for a procedure, such that the corresponding set of the recursion position sets is *optimal*, i.e. it is a *maximal* set of *minimal* recursion position sets.

For instance, {y} is a recursion position set for the procedure `member` which is obtained e.g. from a measure term `count(y)`, where

```
function count(z:sexpr):nat <= ...
```

computes the number of `cons`-cells in `z`. Since termination of `member` cannot be proved with a measure term using only `x`, and {`x,y`} is not minimal, {{y}} is an optimal set of recursion position sets for `member`. Based on the measure term `count(y)`, the termination hypothesis

```
all x,y:sexpr
if(y=cons(car(y),cdr(y)),
    if(x=car(y),true,count(y)>count(cdr(y))),
    true)
```

is generated by ✓eriFun.

For the procedure `minus`, both `x` and `y` are useful measure terms. As {`x,y`} is not minimal, {{x},{y}} is an optimal set of recursion position sets for `minus`. Based on the measure term `x`, the termination hypothesis

```
all x,y:nat
if(x=0,
    true,
    if(y=0,true,x>pred(x)))
```

is generated, and based on the measure term `y`, the termination hypothesis

```
all x,y:nat
if(x=0,
    true,
    if(y=0,true,y>pred(y)))
```

is generated by ✓eriFun. For the procedure

```
function subtract(x,y:nat):bool <=
  if x>y
    then succ(subtract(x,succ(y)))
    else 0
  fi
```

{x,y} is a recursion position set obtained e.g. from the measure term `minus(x,y)`. Since termination of `subtract` cannot be proved with a measure term using only `x` or `y`, {{x,y}} is an optimal set of recursion position sets for `subtract`. Based on the measure term `minus(x,y)`, the termination hypothesis

```
all x,y:nat
if(x>y,minus(x,y)>minus(x,succ(y)),true)
```

is generated by ✓eriFun. For the procedure

```
function gcd(x,y:nat):nat <=
  if x=0
    then y
    else if y=0
           then x
           else if x>y
                  then gcd(minus(x,y),y)
                  else gcd(x,minus(y,x))
                fi
         fi
  fi
```

{x,y} is a recursion position set obtained from the *sequence* `x,y` of measure terms. Since termination of `gcd` cannot be proved with a measure term using only `x` or `y`, {{x,y}} is an optimal set of recursion position sets for `gcd`. Based on the sequence `x,y` of measure terms, the termination hypotheses

```
all x,y:nat
if(x=0,
    true,
    if(y=0,
       true,
       if(x>y,
          if(x>minus(x,y),
            true,
            if(x=minus(x,y),y>y,false)),
          true)))
```

and

```
all x,y:nat
if(x=0,
    true,
    if(y=0,
       true,
       if(x>y,
          true,
          if(x>x,
            true,
            if(x=x,y>minus(y,x),false)))))
```

are generated by ✓eriFun.

**Irrelevant Conditions**

✓`eriFun` automatically generates *induction axioms* from the definition of a procedure and the information obtained by an analysis of the proofs of the termination hypotheses generated for this procedure. A termination hypothesis expresses that the arguments of an *initial procedure call* are greater (wrt. a well-founded relation) than the arguments of a *recursive call*, under the proviso, however, that the *conditions* leading to that recursive call are *satisfied*.

When generating an induction axiom from the proofs of the termination hypotheses, the conditions, which lead to recursive calls, define the *step cases* of the induction (and the base cases then are obtained as the complement of the step cases). As procedures may have conditions, which lead to a recursive call but are *irrelevant* for the procedure's termination, *weak induction axioms* may result if these irrelevant conditions are not disregarded upon the generation of induction axioms. On the contrary, avoiding irrelevant conditions results in *stronger induction axioms*, which then may support subsequent verifications considerably.

For instance, the procedure `minus` and its recursion position set {$x$} suggest the *weak* induction axiom

$$\frac{\begin{array}{ll}\forall x,y{:}\mathrm{nat}\ \ x{=}0 \vee y{=}0 & \rightarrow\ \varphi[x,y] \\ \forall x,y{:}\mathrm{nat}\ \ x{\neq}0 \wedge y{\neq}0 \wedge \forall y'{:}\mathrm{nat}\ \varphi[x/x\text{-}1,\ y/y'] & \rightarrow\ \varphi[x,y]\end{array}}{\forall x,y{:}\mathrm{nat}\ \ \varphi[x,y]\ .}$$

Since the condition „$y{\neq}0$" is irrelevant for the termination of `minus` (wrt. the recursion position set {$x$}), the *stronger* induction axiom

$$\frac{\begin{array}{ll}\forall x,y{:}\mathrm{nat}\ \ x{=}0 & \rightarrow\ \varphi[x,y] \\ \forall x,y{:}\mathrm{nat}\ \ x{\neq}0 \wedge \forall y'{:}\mathrm{nat}\ \varphi[x/x\text{-}1,\ y/y'] & \rightarrow\ \varphi[x,y]\end{array}}{\forall x,y{:}\mathrm{nat}\ \ \varphi[x,y]}$$

can be generated when disregarding the irrelevant condition „$y \neq 0$“. This induction axiom is stronger, because it provides *induction hypotheses* also in the cases $\varphi[n+1, 0]$, whereas these cases must be treated as *base cases* in the weak induction axiom.

✓`eriFun` considers a condition as irrelevant iff this condition is *not used* in the *proof* of a termination hypothesis. This means that the system still succeeds in proving the termination requirement of the procedure also if these conditions are stripped off the preconditions of the termination hypothesis.

For instance,

```
all x,y:nat
if(x=0, true, x>pred(x))
```

obviously is a *true* quantification which also is sufficient for the termination of the procedure `minus` (wrt. the recursion position set $\{x\}$), and therefore „$y \neq 0$“ is an irrelevant condition here.

Irrelevancy always is determined *relative* to a *recursion position set*. If termination of `minus` is proved wrt. the recursion position set $\{y\}$, irrelevancy of „$y \neq 0$“ would mean that

```
all x,y:nat
if(x=0, true, y>pred(y))
```

also is a *true* quantification sufficient for the termination of `minus` (now wrt. the recursion position set $\{y\}$). However, this quantification obviously is *false*, but one obtains „$x \neq 0$“ as an irrelevant condition of `minus` wrt. the recursion position set $\{y\}$.

The determination of irrelevant conditions is incorporated in ✓`eriFun`'s automated termination analysis. However, the routine is not foolproof. It may happen that the system regards *irrelevant* conditions as *relevant* because these conditions are used in a proof, although this is not required. Therefore also the user should feel responsible for the generation of strong induction axioms, by inspecting the proofs of termination hypotheses and editing these proofs if required in order to avoid the use of irrelevant conditions.

***Automated Termination Analysis***

✓eriFun's automated termination analysis

- *computes* measure terms, and
- *verifies* the termination hypotheses based on these measure terms.

However, as the problem to verify the termination of procedures is not semi-decidable, any automated termination analysis *must* fail for some terminating procedures. So it may happen that for a user defined procedure, the system

*(i)* fails to compute measure terms, or

*(ii)* computes measure terms yielding a non-optimal set of recursion position sets, or

*(iii)* computes measure terms, but regards irrelevant conditions in some of the generated termination hypotheses as relevant, or

*(iv)* computes measure terms but fails to prove one of the generated termination hypotheses at least.

In cases *(i)* and *(ii)*, the user must step in to provide the appropriate (sequence of) measure terms, in case *(iii)* the user has to eliminate the irrelevant conditions and in case *(iv)* the user is asked to edit the proofs of the unproven termination hypotheses.

**Argument-bounded Procedures**

✓eriFun's automated termination analysis is based on so-called *size-measures*: For each *recursively* defined data structure `sort` from the signature, $\#_{\texttt{sort}}(\texttt{q})$ denotes the *size* of a value `q` which is defined as the number of occurrences of *reflexive* constructors `cons` : ... × `sort` × ... → `sort` in `q`. E.g. $\#_{\texttt{nat}}$ simply computes the natural number represented by a `nat`-value `q`, and $\#_{\texttt{sexpr}}$ counts the number of `cons`-cells in an `sexpr`-value `q`.

The key concept for proving termination automatically is that of a *p-bounded procedure*: A *terminating* procedure defined by

$$\texttt{function PROC(}x_1\texttt{:sort}_1\texttt{,...,}x_n\texttt{:sort}_n\texttt{):sort <= ...}$$

is *p-bounded* iff each procedure call $\texttt{PROC(}q_1\texttt{,...,}q_p\texttt{,...,}q_n\texttt{)}$ (where $q_1,...,q_n \in \mathcal{V}al$) returns a value whose size is less than or equal to the size of its actual parameter $q_p$ at argument position *p*. `PROC` is *argument-bounded* iff it is *p*-bounded for some argument position *p*.

Formally expressed, `PROC` is *p*-bounded iff for all $q_1,...,q_n \in \mathcal{V}al$

$$\#_{\texttt{sort}}(q_p) \geq \#_{\texttt{sort}}(eval(\texttt{PROC(}q_1\texttt{, ... ,}q_p\texttt{, ... ,}q_n\texttt{)})) \ .$$

**Difference-Procedures**

Each *p*-bounded procedure PROC is assigned a so-called (terminating) *p*-*difference-procedure*

$$\texttt{function PROC\$p(}x_1\texttt{:sort}_1\texttt{, …,}x_n\texttt{:sort}_n\texttt{):bool <= ...}$$

which provides a requirement that a procedure call $\texttt{PROC(}q_1\texttt{, … ,}q_p\texttt{, …,}\ q_n\texttt{)}$ returns a value whose size is *strictly less* than the size of its actual parameter $q_p$.

The relation between a *p*-bounded procedure PROC and its *p*-difference-procedure PROC$p is formally expressed by

$$\mathit{eval}(\texttt{PROC\$p(}q_1\texttt{, … ,}q_n\texttt{)}) = \mathit{true}$$

$$\rightarrow$$

$$\#_{\texttt{sort}}(q_p) \ > \ \#_{\texttt{sort}}(\mathit{eval}(\texttt{PROC(}q_1\texttt{, … ,}q_p\texttt{, … ,}q_n\texttt{)}))$$

for all $q_1,\ldots,q_n \in \mathcal{V}al$.

After the termination of a procedure PROC has been verified, the system decides whether this procedure is *p*-bounded. If so, the system synthesizes a (terminating) *p*-difference-procedure PROC$p and inserts PROC$p into the actual program. As a procedure may be *p*-bounded for more than one argument position *p*, several difference-procedures may be synthesized for PROC and inserted into the actual program.

Since the *1*-boundedness requirement always holds for *reflexive* selectors of data structures, the system also synthesizes a (non-recursively defined) *1*-difference-procedure SEL$1 for each reflexive selector SEL and inserts SEL$1 into the actual program.

**Synthesis of Termination Hypotheses**

The recognition of *p*-bounded procedures and the synthesis of *p*-difference-procedures is utilized for generating the termination hypotheses of procedures which *use* *p*-bounded procedures in *recursive calls*. If a procedure, say `FOO`, is recursively defined using a procedure call of a *p*-bounded procedure `PROC` in a recursive call, a termination hypothesis is generated for `FOO` using the *p*-difference-procedure `PROC$p` instead of the predefined procedure `>`.

For instance, after the insertion of the data structure `sexpr`, the system synthesizes the *1*-difference-procedures

```
function car$1(x:sexpr):bool <=
  x=cons(car(x),cdr(x))
```

and

```
function cdr$1(x:sexpr):bool <=
  x=cons(car(x),cdr(x))
```

for the reflexive selectors `car` and `cdr` of `sexpr`. Utilizing its knowledge about the *1*-boundedness of `cdr` and the *1*-difference procedure `cdr$1`, the system now automatically verifies the termination hypotheses

```
all x,y:sexpr
if(y=cons(car(y),cdr(y)),
   if(x=car(y),
      true,
      cdr$1(y)),
   true)
```

automatically generated for the procedure `member`.

Having verified the termination of `member`, ✓`eriFun` now recognizes that `member` is *2*-bounded and therefore synthesizes the *2*-difference-procedure

```
function member$2(x,y:sexpr):bool <=
  if y=cons(car(y),cdr(y))
    then if x=car(y)
           then false
           else true
         fi
    else false
  fi .
```

Also, after the termination of minus has been verified, ✓eriFun recognizes that minus is *1*-bounded and therefore synthesizes the *1*-difference-procedure

```
function minus$1(x,y:nat):bool <=
  if x=0
    then false
    else if y=0
           then false
           else true
         fi
  fi .
```

Utilizing its knowledge about the *1*-boundedness of minus and the *1*-difference procedure minus$1, the system now automatically verifies the termination hypotheses

```
all x,y:nat
if(x=0,
   true,
   if(y=0,
      true,
      if(x>y,minus$1(x,y),true)))
```

and

```
all x,y:nat
if(x=0,
   true,
   if(y=0,
      true,
      if(x>y,true,minus$1(y,x))))
```

automatically generated for the procedure gcd.

**Optimized Difference-Procedures**

Since difference-procedures are used in termination proofs, theorem proving is supported if difference-procedures are as *simple as possible*. For each recursive call in a difference-procedure, ✓`eriFun` generates a *pair* of recursion elimination formulas, one formula which is sufficient to replace the recursive call by `true` and the other formula sufficient to replace the recursive call by `false`. A difference-procedure is *accepted* iff the user explicitly accepts the definition or otherwise for each pair of recursion elimination formulas either one is *verified* or both are *falsified*.

✓`eriFun` attempts to *optimize* each difference-procedure `PROC$p` by

- *simplification of the body* of `PROC$p`, and
- *elimination of recursive calls* in the body of `PROC$p`.

However, such optimization steps may be insufficient or even fail. Since the presence of *optimized* difference-procedures may strongly support subsequent verification tasks, the user should *immediately* step in in such a case

- to support simplification of the procedure body and also of recursion elimination by providing appropriate *verified* lemmas (before definition of `PROC`), and
- to *edit* a *proof* of a *recursion elimination formula* which the system has generated for a recursive call in a difference-procedure (in case such a formula is *true*), or otherwise
- to *edit* a *refutation* of a *recursion elimination formula*.

**Boundedness-Lemmas**

After insertion of a *p*-bounded procedure

```
function PROC(x1:sort1,…,xn:sortn):nat <= ...
```

into the actual program, the system generates a so-called *boundedness-lemma*

```
lemma PROC$p_bounded <= all x1:sort1,…,xn:sortn
    if(PROC(x1,…,xn)>xp,
       false,
       if(PROC$p(x1,…,xn),
         xp>PROC(x1,…,xn),
         PROC(x1,…,xn)=xp))
```

which is also inserted into the actual program. A boundedness-lemma is based on the fact that `q=r` iff $\#_{\texttt{nat}}(\texttt{q}) = \#_{\texttt{nat}}(\texttt{r})$ for each pair of `nat`-values `q` and `r`, hence reasoning about the *p*-boundedness of `PROC` can be explicitly represented by a lemma to be available also for subsequent verification tasks not related to termination. A boundedness-lemma expresses that

- $\texttt{PROC(q}_1\texttt{,...,q}_p\texttt{,...,q}_n\texttt{)} \not> \texttt{q}_p$ ,
- $\texttt{q}_p\texttt{>PROC(q}_1\texttt{,...,q}_p\texttt{,...,q}_n\texttt{)}$ , if $\texttt{PROC\$p(q}_1\texttt{,...,q}_p\texttt{,...,q}_n\texttt{)}$ is *true*, and
- $\texttt{q}_p\texttt{=PROC(q}_1\texttt{,...,q}_p\texttt{,...,q}_n\texttt{)}$ , if $\texttt{PROC\$p(q}_1\texttt{,...,q}_p\texttt{,...,q}_n\texttt{)}$ is *false*

for all `nat`-values $\texttt{q}_1\texttt{,...,q}_p\texttt{,...,q}_n$.

For instance, upon insertion of the *1*-bounded procedure `minus`, the boundedness-lemma

```
lemma minus$1_bounded <= all x:nat,y:nat
    if(minus(x,y)>x,
       false,
       if(minus$1(x,y),
         x>minus(x,y),
         minus(x,y)=x))
```

is also inserted into the actual program.

**Projection-Lemmas**

After insertion of a *p*-bounded procedure

$$\texttt{function PROC(}x_1\texttt{:}sort_1\texttt{,}\ldots\texttt{,}x_n\texttt{:}sort_n\texttt{):sort <= ...}$$

with `sort` ≠ `nat` into the actual program, the system generates a so-called *projection-lemma*

$$\begin{aligned}&\texttt{lemma PROC\$p_projection <= all } x_1\texttt{:}sort_1\texttt{,}\ldots\texttt{,}x_n\texttt{:}sort_n\\&\quad\texttt{if(PROC\$p(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{),true,PROC(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)=}x_p\texttt{)}\end{aligned}$$

which is also inserted into the actual program. Similar to a boundedness-lemma, reasoning about the *p*-boundedness of `PROC` is explicitly represented by a projection-lemma to be available also for subsequent verification tasks not related to termination. A projection-lemma expresses that each procedure call $\texttt{PROC(}q_1\texttt{,...,}q_p\texttt{,...,}q_n\texttt{)}$ returns $q_p$ whenever the call $\texttt{PROC\$p(}q_1\texttt{,...,}q_p\texttt{,...,}q_n\texttt{)}$ of the corresponding difference procedure returns *false*.

Projection-lemmas are not generated for difference-procedures based on *selectors*. A projection-lemma is *not* necessarily *true* as in general only

$$\#_{\texttt{sort}}(\mathit{eval}(\texttt{PROC(}q_1\texttt{,}\ldots\texttt{,}q_p\texttt{,}\ldots\texttt{,}q_n\texttt{}))) = \#_{\texttt{sort}}(q_p)$$

holds, if $\mathit{eval}(\texttt{PROC\$p(}q_1\texttt{,}\ldots\texttt{,}q_n\texttt{}))$=*false*, instead of the stronger requirement

$$\mathit{eval}(\texttt{PROC(}q_1\texttt{,}\ldots\texttt{,}q_p\texttt{,}\ldots\texttt{,}q_n\texttt{}))) = q_p$$

expressed by the projection-lemma.

For instance, the projection-lemma `member$2_projection` is obviously *false*, because $\mathit{eval}(\texttt{member\$2(}\ldots\texttt{, atom(0)}))) = \mathit{false}$ but $\mathit{eval}(\texttt{member(}\ldots\texttt{,atom(0)}))) = \texttt{nil} \neq \texttt{atom(0)}$.

Since the presence of *verified* projection-lemmas may strongly support subsequent verification tasks, the user should *immediately* step in after the generation of a projection-lemma

- to *edit a proof* of the projection-lemma, or otherwise
- to *edit a refutation* of the projection-lemma (simply to be sure, that no proof exists).

<>

# File

## About File

The *File Menu* provides all commands

- to reset the system state,
- to load and store data hold in files,
- to alter the system parameters, and
- to terminate the system.

Commands are selected from the menu bar or from the context menu.

✓`eriFun` supports two file types, viz. *vf*-files and *fp*-files:

*vf*-files store the whole system state, i.e. the state which results from the user definitions of a program plus the prooftrees of the lemmas, the termination hypotheses and the recursion elimination formulas in this program. *vf*-files can be created and processed by ✓`eriFun` only. *vf*-files carry the extension `.vf` and are the primary input/output medium of ✓`eriFun`.

*fp*-files store the user definitions of a program (plus some technical data for maintaining the folder structure of the program) in plain ASCII. All prooftrees in this program are *skipped* when writing *fp*-files. *fp*-files can be created and processed by ✓`eriFun` or alternatively by each conventional ASCII-editor. *fp*-files carry the extension `.fp` and are used for system development. Users may create *fp*-files to look at a program as a whole in some editor of their choice.

For files of both types so-called *backup files* may exist. Backup files carry the extension `.vf.bak` or `.fp.bak` respectively and are used to backtrack to former file contents. All commands of the *File Menu* which are related to *vf*- or *fp*-files may be also applied to files with extension `.vf.bak` or `.fp.bak` respectively.

<>

### New

Resets ✓eriFun to the *initial state*, i.e. the state the system has immediately after system start.

### Open

Resets the system state to the state the system had when the opened *vf*-file was created.

**Note 1** The *vf*-file is not opened and a message „*Incompatible! This file is in VeriFun* nnn *format.*“ results instead if the *vf*-file to be opened was not created by the ✓eriFun version under execution and cannot be processed by this version.

A message „*This file is in VeriFun* nnn *format.*“ results if the *vf*-file to be opened was not created by the ✓eriFun version under execution but can be processed by this version.

### Save

Saves the system state to the *vf*-file which has been used in the most recent *Open*, *Save* or *Save as* command. The former file content is retained in a file with extension *.bak*.

### Save as

Saves the system state to a user specified *vf*-file.

### *Read*

Reads an *fp*-file and applies the *Insert* command of the *Program Menu* to all program elements of the read *fp*-file.

**Note 1** A warning „*`Unproven termination hypothesis generated for proc`*“ may result when reading an *fp*-file. Use *Set Termination* of the *Program Menu* and continue with editing a proof (or a refutation) of such a termination hypothesis, because such an edit is not possible after this procedure is subsequently used in a prooftree.

### *Write*

Writes the actual program (without considering prooftrees) to the *fp*-file which has been used in the most recent *Read*, *Write* or *Write as* command. The former file content is retained in a file with extension *.bak*.

### *Write as*

Writes the actual program (without considering prooftrees) to a user specified *fp*-file.

## *Import*

Opens the *Import Window* to allow the import of program items and prooftrees stored in an *vf*- or *fp*-file into the actual program.

The actual program is displayed in the *Import Window* as the **Import Program**. Each program stored in an *vf*- or *fp*-file can be used as an **Export Program**.

To **define an Export Program**, in the *Import Window* use
⇨ the **Open** button to define an **Export Program** stored in an *vf*-file,
⇨ the **Read** button to define an **Export Program** stored in an *fp*-file.

To **import a program item and its prooftrees** from the **Export Program** into the **Import Program**, in the *Import Window*
⇨ select an item in the **Import Program** to define the place where the imported item should be located,
⇨ select the item to be imported in the **Export Program**,
⇨ push the **Import** button of the *Import Window* or alternatively press the right mouse button and select **Import** from the context menu,
⇨ continue with further imports, if required,
⇨ push the **Close** button to quit.

To **copy a program element from the Export Program into the clipboard**
⇨ select the program element in the **Export Program**,
⇨ press the right mouse button and select **Copy** from the context menu.

To **open the Program Property Window of a program item of the Export Program**
⇨ select the program item in the **Export Program**,
⇨ press the right mouse button and select **Properties** from the context menu.

**Note 1** Only program items from the *Export Program*, which are *compatible* with the *Import Program* can be selected for import. A program *element* from the *Export Program* is *compatible* with the *Import Program* iff

- ⇨ no program element from the *Import Program* shares the same name with the program element of the *Export Program* or the *body* of a program element from the *Import Program* and the program element of the *Export Program* coincides, if both program elements share the same *name*,
- ⇨ the *status* of a program element from the *Import Program* is less than or equal to the *status* of the program element from the *Export Program*, if both program elements share the same *name* and coincide in the *bodies*,
- ⇨ a verified procedure from the *Import Program* has the same set of recursion position sets as the procedure from the *Export Program*, if both procedures share the same *name* and coincide in the *bodies*,
- ⇨ the program element from the *Export Program* is an argument-bounded procedure for which a projection-lemma exists in the *Export Program*, whenever this projection-lemma also is present in the *Import Program*, and
- ⇨ all other program elements of the *Export Program*, on which the program element selected for import depends, are also compatible with the *Import Program*.

A program *folder* from the *Export Program* is *compatible* with the *Import Program* iff all program items in this folder are compatible with the *Import Program*.

**Note 2** The following items of the *Export Program* cannot be selected for import:

- ⇨ Predefined program items,
- ⇨ system generated program elements,
- ⇨ the root program folder, and
- ⇨ a program folder, the items of which are distributed over *several* folders of the *Import Program*.

**Hint 1** In the *Import Window*,
⇨ all names of the program items in the *Export Program*, which are compatible with the *Import Program*, are displayed *light* or *normal*,
⇨ the names of all incompatible program items are displayed *red*,
⇨ the names of all program items of the *Export Program*, which are already included in the *Import Program* are displayed *light*.

**Hint 2** Importing a program element from the *Export Program* also causes the import of all other program elements from the *Export Program*
⇨ on which the program element selected for import depends,
⇨ which had been generated by the system for the program element selected for import.

To **recognize the program elements which are imported in addition**
⇨ select the program element to be imported in the **Export Program**,
⇨ press the right mouse button and select **Properties** from the context menu to open the **Program Property Window** of the selected element,
⇨ select the **Usage** tab in the *Program Property Window* to see the program elements which are imported in addition in the **Uses**-column.

**Hint 3** Procedures and lemmas from the *Export Program*, which are already included in the *Import Program*, may have different (termination) prooftrees in the *Export-* and in the *Import Program*. The import of those program elements replace the prooftrees in the *Import Program* by the prooftrees of the *Export Program*.

**Hint 4** By pressing the `ctrl`- or the `shift`-key, several program items may be selected for a joint import. While other work may progress when the *Import Window* is open, it is recommended to close this window if no further imports are wanted in order to save the time the system needs for testing compatibility.

**Hint 5** Use the *Rename* command from the *Program Menu* to resolve incompatibilities between items of an *Import-* and an *Export Program* caused by name clashes.

***(Customize)***

Opens a dialog to alter

- ⇨ the parameters which control the *Symbolic Evaluator*,
- ⇨ the flags to enable/disable the *automated termination analysis*,
- ⇨ the flag to enable/disable the *lemma-filter-heuristic*,
- ⇨ the flag to enable/disable the *assumption-heuristic*,
- ⇨ the flag to enable/disable the *next-rule-heuristic* and the automated application of induction hypotheses.

**Note 1** The data in the *Customize Window* can be modified for system development only.

***User Settings***

Opens a dialog to switch on/off the applicability test for the computed rules.

**Hint 1** Navigation in prooftrees of large programs may be speeded up by disabling the tests for the applicability of the computed rules.

***Exit***

Ends the execution of ✓eriFun.

# Program

## About Program

The *Program Menu* provides all commands
- to edit a program,
- to guide the generation of termination hypotheses,
- to open a prooftree in the *Proof Window*, and
- to initiate a verification.

Commands are selected from the menu bar or from the context menu.

***Program Elements, Folders and Items***
Data structures, procedures and lemmas of a program are called *program elements*. Program elements may be collected in *program folders*, which may also contain other program folders. Program folders are referred to by a *name*. Program elements and program folders are called *program items*. User created program items may carry a *comment*, given as an arbitrary sequence of ASCII characters.

***The Actual Program***
Program items are listed in the *Program Window*. The set of all program items listed in this window is referred to as the *actual program*. All items of the actual program carry *distinct names*. Using the *drag&drop*-feature, program items may be rearranged in the actual program and moved to or out off program folders at the user's wish. These arrangements are irrelevant for verification and for the system's ✓`afp`-language parser.

***Predefined Program Items***

✓eriFun is in *initial state* just after system start or after execution of the *New* command from the *File Menu*. Being in initial state, the actual program is a verified program and consists of the *root program folder* `Program` which only contains the program folder `Predefined`.

The program folder `Predefined` contains

- the data structure `bool` which defines *truth values*,
- the data structure `nat` which defines the *set of natural numbers*,
- the procedure `>` which defines the usual >-*relation* on natural numbers, and
- certain lemmas about these program elements.

The data structure `bool` is predefined because all statements about a program are expressed with *boolean* terms in ✓eriFun. The data structure `nat` and the procedure `>` are predefined because in ✓eriFun all hand-crafted termination proofs are based on the (well-founded) >-relation. The lemmas in the program folder `Predefined` are predefined because they formulate facts which are particularly useful for proving the termination of procedures if ✓eriFun's automated termination analysis fails.

***System Generated Program Elements***
Upon insertion of a user defined program element into the actual program, the system may generate additional program elements which are also inserted into the actual program:

- A lemma `struct_=_transitivity`, called a *transitivity-lemma*, is inserted upon insertion of a data structure `struct`.
- A difference-procedure `sel$1` is inserted upon insertion of a data structure `struct` with a reflexive selector `sel`.
- A difference-procedure `proc$`$N$ is inserted upon insertion of an $N$-bounded procedure `proc`.
- A boundedness-lemma `proc$`$N$`_bounded` is inserted upon insertion of an $N$-bounded procedure `proc` with rangesort `nat`.
- A projection-lemma `proc$`$N$`_projection` is inserted upon insertion of an $N$-bounded procedure `proc` with a rangesort different from `nat`.

***Status of Program Elements***
Each element of the actual program, each termination hypothesis and each recursion elimination formula is assigned a *status* to indicate the development of a proof for that element or formula so far. The status of a program element is displayed in the *Program Window* by a color according to the following assignments:

The status of a termination hypothesis is displayed in the *Termination Window* and the status of a recursion elimination formula is displayed in the *Synthesis Details Window* using the same color assignments.

*Data structures* have always status *verified*.

A user defined *lemma*, a *termination hypothesis* or a *recursion elimination formula* has status

- *ignored* iff the body of the lemma contains a call of a procedure with a status different from *verified*, and otherwise
- *falsified* iff it has been refuted,
- *verified* iff the system holds a finished prooftree for the closed quantification,
- *developed* iff the system holds a closed prooftree for the closed quantification using no *falsified* lemmas and some non-*verified* lemma,
- *ready* in the remaining cases.

*Predefined program elements* and *transitivity-lemmas* have always status *verified*.

A *boundedness-lemma* has status

- *ignored* iff the associated difference-procedure has a status different from *verified*, and otherwise
- *verified*.

A *projection-lemma* has status

- *ignored* iff the associated difference-procedure has a status different from *verified*, and otherwise
- *falsified* iff it has been refuted,
- *verified* iff the system holds a finished prooftree for the closed quantification,
- *developed* iff the system holds a closed prooftree for the closed quantification using no *falsified* lemmas and some non-*verified* lemma,
- *ready* in the remaining cases.

A user defined *procedure* has status

- *ignored* iff
  - ⇨ the body of the procedure contains a call of another procedure with a status different from *verified*, or
  - ⇨ the procedure is recursively defined but no termination hypothesis has been generated for it so far, or
  - ⇨ a termination hypothesis generated for the procedure contains a call of a difference-procedure with a status different from *verified*,
- *verified* iff
  - ⇨ the procedure is undefined, or
  - ⇨ the procedure is non-recursively defined and its body contains only calls of *verified* procedures, or
  - ⇨ the procedure is recursively defined, its body contains only (non-recursive) calls of *verified* procedures, and the procedure possesses a recursion position set such that all termination hypotheses belonging to that set are *verified*,
- *developed* iff the procedure is recursively defined but not *verified*, its body contains only (non-recursive) calls of *verified* procedures, and the procedure possesses a recursion position set such that all termination hypotheses belonging to that set are *verified* or *developed* and some termination hypothesis is *developed*,
- *ready* in the remaining cases.

A *difference-procedure* has status

- *verified* iff
  - ⇨ the procedure has only pairs of recursion elimination formulas such that either one of the pair is *verified* or both are *falsified*, or
  - ⇨ the procedure is explicitly accepted by the user,
- *ready* in the remaining cases.

The subset of all *verified* program elements of the actual program is called the *verified program*. The verified program consists of all program elements listed in the *Program Window* with green color.

Termination hypotheses, recursion elimination formulas, non-*ignored* user defined lemmas and projection-lemmas are assigned a *prooftree*. The status of such a quantification is inherited from the status of the root nodes of the prooftree which is assigned to this quantification.

Consequently, such a quantification has status

- *falsified* iff the root node of the prooftree for the quantification has status *disproved*,
- *verified* iff the root node of the prooftree for the quantification has status *proved*,
- *developed* iff the root node for the quantification has status *completed*,
- *ready* iff the root node of the prooftree for the quantification has status *unproved* or *unprovable*.

The status set for program elements is *partially ordered* by

$$\textit{ignored} < \textit{ready} < \textit{developed} < \textit{verified}, \textit{falsified}.$$

*Program folders* are also displayed with *colors* in the *Program Window*: A program folder inherits the color of its members iff all members share the same color, and is colored *light blue* otherwise.

***Symmetry and Commutativity***
A lemma like

```
lemma perm_symmetric <= all x,y:list
  if(perm(x,y),perm(y,x),true)
```

is called a *symmetry lemma*, and `perm` is a *symmetric function symbol* iff `perm_symmetric` is verified.

A lemma like

```
lemma plus_commutative <= all x,y:nat
  plus(x,y)=plus(y,x)
```

is called a *commutativity lemma*, and `plus` is a *commutative function symbol* iff `plus_commutative` is verified.

Symmetric and commutative function symbols are recognized by the *Symbolic Evaluator* and by some proof rules of the *Proof Rules* submenu of the *Proof* menu.

<><><>

## *New Folder*

Opens a dialog to insert a user specified name in the *Name* field for the program folder to be created. Comments for this folder may be inserted into the *Comment* field.

**Hint 1** The new folder is located at the bottom of the program folder which is selected in the *Program Window* when issuing the *New Folder* command.

The new folder is located at the bottom of the program folder holding the program element which is selected in the *Program Window* when issuing the *New Folder* command.

## *Insert*

Opens the *Insert Dialog* for editing a new program element to be inserted into the actual program. The program element is edited in the *Element* field of the *Insert Dialog*, comments may be inserted into the *Comment* field.

To **edit a new program element**, in the *Insert Dialog* use
⇨ the **text buttons** for frequently used input fragments,
⇨ the **Clear** button to clear the **Element** field of the *Insert Dialog*,
⇨ the **Cancel** button to withdraw,
⇨ the **OK** button to quit.

**Note 1** A warning „*`Unproven termination hypothesis generated for proc`*“ may result upon insertion of a procedure `proc`. Use *Set Termination* of the *Program Menu* and continue with editing a proof (or a refutation) of such a termination hypothesis, because such an edit is not possible after this procedure is used subsequently in a prooftree.

**Hint 1** Upon insertion of a new program element, system generated program elements may be also inserted into the actual program and the Symbolic Evaluator may be called to prove termination hypotheses and/or recursion elimination formulas.

**Hint 2** In order to make difference-procedures and projection-lemmas immediately available for subsequent verification tasks, it is recommend upon insertion of an *N*-bounded procedure `proc` to continue with
⇨ editing a proof (or a refutation) of a *recursion elimination formula*, if non-*verified* recursion elimination formulas have been generated for the difference-procedure `proc$`*N*,
⇨ editing a proof (or a refutation) of a projection lemma generated for the difference-procedure `proc$`*N*.

## *Duplicate*

Opens the *Insert Dialog* for editing a new program element to be inserted into the actual program. The program element is edited in the *Element* field of the *Insert Dialog*, comments may be inserted into the *Comment* field.

The *Element* field of the *Insert Dialog* contains the definition of the program element which was selected when issuing the *Duplicate* command. The *Comment* field holds the comment of this program element.

To **edit a duplicated program element**, in the *Insert Dialog* use
⇨ the **text buttons** for frequently used input fragments,
⇨ the **Clear** button to clear the **Element** field of the *Insert Dialog*,
⇨ the **Cancel** button to withdraw,
⇨ the **OK** button to quit.

**Note 1** A warning *„Unproven termination hypothesis generated for proc“* may result upon insertion of a procedure `proc`. Use *Set Termination* of the *Program Menu* and continue with editing a proof (or a refutation) of such a termination hypothesis, because such an edit is not possible after this procedure is used subsequently in a prooftree.

**Hint 1** Upon insertion of a new program element, system generated program elements may be also inserted into the actual program and the Symbolic Evaluator may be called to prove termination hypotheses and/or recursion elimination formulas.

**Hint 2** In order to make difference-procedures and projection-lemmas immediately available for subsequent verification tasks, it is recommend upon insertion of an $N$-bounded procedure `proc` to continue with

- ⇨ editing a proof (or a refutation) of a *recursion elimination formula*, if non-*verified* recursion elimination formulas have been generated for the difference-procedure `proc$`$N$,
- ⇨ editing a proof (or a refutation) of a projection lemma generated for the difference-procedure `proc$`$N$.

**Hint 3** *Duplicate* eases edits of program elements obtained from other program elements by slight alterations.

## *Rename*

Opens a dialog to provide a new name for the selected program item. New names for *constructor*- and *selector*-functions of a data structure are provided by using this command with the appropriate data structure selected.

**Note 1** *Rename* does not apply to
⇨ predefined program items except the root program folder `Program`,
⇨ system generated program elements.

**Hint 1** *Rename* is particularly useful for resolving incompatibilities between items of an *Import*- and an *Export Program* caused by name clashes.

## *Comment*

Opens an dialog to create or modify a comment for the selected program item.

**Note 1** *Comment* does not apply to
⇨ predefined program items except the root program folder `Program`,
⇨ system generated program elements except difference-procedures.

## *Modify*

Opens the *Insert Dialog* for modifying the selected program element. The program element is edited in the *Element* field of the *Insert Dialog*, comments may be inserted into the *Comment* field.

The *Element* field of the *Insert Dialog* contains the definition of the program element which was selected when issuing the *Modify* command. The *Comment* field holds the comment of this program element.

To **modify a program element**, in the *Insert Dialog* use
⇨ the **text buttons** for frequently used input fragments,
⇨ the **Clear** button to clear the **Element** field of the *Insert Dialog*,
⇨ the **Cancel** button to withdraw,
⇨ the **OK** button to quit.

**Note 1** A warning „*Unproven termination hypothesis generated for proc*“ may result upon insertion of a procedure `proc`. Use *Set Termination* of the *Program Menu* and continue with editing a proof (or a refutation) of such a termination hypothesis, because such an edit is not possible after this procedure is used subsequently in a prooftree.

**Note 2** All prooftrees assigned to the modified program element either are *pruned at their root* or are *destroyed*.

**Note 3** *Modify* does not apply to
⇨ predefined program elements,
⇨ system generated program elements,
⇨ a program element which is used by another program element or in a prooftree.

**Hint 1** All program elements, which *use* a program element, are listed in the **Used by**-column of the **Usage** tab in the Program Viewer or the program element's Program Property Window.

**Hint 2** Upon insertion of a new program element, system generated program elements may be also inserted into the actual program and the Symbolic Evaluator may be called to prove termination hypotheses and/or recursion elimination formulas.

**Hint 3** In order to make difference-procedures and projection-lemmas immediately available for subsequent verification tasks, it is recommend upon insertion of an $N$-bounded procedure `proc` to continue with

- ⇨ editing a proof (or a refutation) of a *recursion elimination formula*, if non-*verified* recursion elimination formulas have been generated for the difference-procedure `proc$`$N$,
- ⇨ editing a proof (or a refutation) of a projection lemma generated for the difference-procedure `proc$`$N$.

## *Delete*

Deletes the selected program item from the actual program. If a program folder is selected, all items which are contained in this folder are deleted too.

**Note 1** *Delete* does not apply to
- ⇨ predefined program items,
- ⇨ system generated program elements except *projection-lemmas*,
- ⇨ a program element which is used by other program elements,
- ⇨ a program folder containing a program element which is used by other program elements.

**Hint 1** All program elements, which use a program element, are listed in the **Used by**-column of the **Usage** tab in the Program Viewer or the program element's Program Property Window.

**Hint 2** Since ✓eriFun does not provide an undo-feature, the system asks for confirmation before deleting the selected program item.

## *Delete depending*

Deletes the selected program element and all program elements which use the selected program element from the actual program. If a program folder is selected, all program items contained in this folder are deleted too.

**Note 1** *Delete depending* does not apply to
⇨ predefined program items,
⇨ system generated program elements except *projection-lemmas*,
⇨ a program item which can be deleted using the *Delete* command.

**Hint 1** All program elements, which use a program element, are listed in the **Used by**-column of the **Usage** tab in the Program Viewer or the program element's Program Property Window.

**Hint 2** Since ✓eriFun does not provide an undo-feature, the system asks for confirmation before deleting the selected program item.

## *Set Termination*

Opens the *Termination Window* for providing (sequences of) measure terms to guide the generation of termination hypotheses of the *procedure* selected when issuing the *Set Termination* command.
/ Opens the *Synthesis Details Window* of the difference-procedure selected when issuing the *Set Termination* command.

To **provide (sequences of) measure terms** in the *Termination Window*

- ⇨ insert a *measure term* or a non-empty *sequence of measure terms* separated by commas in the **Measure Terms Input** field of the *Termination Window*,
- ⇨ see the **Recursion Position Sets/Available** field of the *Termination Window* for a set of variables which must be used when defining (sequences of) *measure terms*,
- ⇨ use the **Clear** button to clear the **Measure Terms Input** field, if required for an insufficient content,
- ⇨ use the **Accept** button to move a (sequence of) *measure term(s)* from the **Measure Terms Input** field to the **Chosen Measure Terms** field in order to submit the data in **Measure Terms Input** field to the system's termination-hypotheses-generation routine,
- ⇨ use the **Remove** button to withdraw a selected (sequence of) *measure term(s)* from the **Chosen Measure Terms** field, if required for an unwanted content,
- ⇨ use the **Modify** button to move a selected (sequence of) *measure term(s)* from the **Chosen Measure Terms** field back to the **Measure Terms Input** field, if modifications are required,
- ⇨ use the **Duplicate** button to insert a copy of a (sequence of) *measure term(s)* selected in the **Chosen Measure Terms** field into the **Measure Terms Input** field, in order to ease the edit of another (sequence of) *measure term(s)*.

and

⇨ push the **Recompute** button to replace all user provided input with the data computed by ✓`eriFun`'s automated termination analysis,
⇨ push the **Proof** button to **open in the Proof Window the prooftree of the termination hypothesis** which is selected in the **Termination Hypotheses** field,
⇨ push the **OK** button to quit.

To **recognize conditions regarded as relevant** in a proof of a termination hypothesis
⇨ select the termination hypothesis in the **Termination Hypotheses** field of the *Termination Window*,
⇨ push the **Proof** button to open in the **Proof Window** the prooftree of the termination hypothesis just selected,
⇨ select the *root node* of the termination hypothesis' prooftree and open the **Proof Property Window** or alternatively open the **Proof Viewer**,
⇨ choose the **Hypotheses** tab of the *Proof Property Window* or of the *Proof Viewer* to see the conditions regarded as *relevant* displayed in *red*.

To **eliminate irrelevant conditions regarded as relevant** in a proof of a termination hypothesis
⇨ prune the prooftree of the termination hypothesis at the root node using the *Prune* command from the *Proof Menu*,
⇨ start the *Delete Hypotheses* command of the *Proof Rules Submenu*, select all irrelevant conditions in the command's dialog and push the **OK** button.

To **accept the selected difference-procedure without (further) optimization**
⇨ push the **Accept** button in the *Synthesis Details Window*.

To **edit a prooftree of a recursion elimination formula** generated for the selected difference-procedure,
⇨ choose the **Recursion Elimination** tab in the *Synthesis Details Window*,
⇨ select a *recursion elimination formula* from the **TRUE-Case** or from the **FALSE-Case** field of the *Synthesis Details Window*,
⇨ push the **Proof** button in the *Synthesis Details Window* to open in the *Proof Window* the prooftree of the selected formula,
⇨ push the **Close** button in the *Synthesis Details Window* to close this window.

**Note 1** *Set Termination* does not apply to
⇨ predefined program elements,
⇨ system generated program elements except difference-procedures,
⇨ non-recursively defined procedures,
⇨ difference-procedures for which no recursion elimination formula has been generated,
⇨ procedures which are used in prooftrees.

**Hint 1** All program elements which use a procedure are listed in the **Used by**-column of the **Usage** tab in the Program Viewer or in the procedure's Program Property Window.

**Hint 2** *Measure terms* computed by ✓`eriFun`'s automated termination analysis are prefixed with the '$' symbol.

**Hint 3** The set of termination hypotheses generated for the (sequence of the) *measure term(s)* selected in the **Chosen Measure Terms** field is displayed in the **Termination Hypotheses** field of the *Termination Window*, where a *termination hypothesis* is assigned a color depending on its status, except that a termination hypothesis is colored *orange*, if it either *can* be refuted or otherwise belongs to a non-executable recursive call.

**Hint 4** A termination hypothesis selected in the **Termination Hypotheses** field is displayed in *pretty print* in the bottom-right field of the *Termination Window*, if the **Termination Hypothesis** tab of this field is chosen. The recursive call corresponding to the selected termination hypothesis is highlighted in top-right field of the *Termination Window*.

**Hint 5** The relation description, which belongs to the (sequence of the) *measure term(s)* selected in the **Chosen Measure Terms** field is displayed in the bottom-right field of the *Termination Window*, if the **Relation Description** tab of this field is chosen.

**Hint 6** Pushing the **Show Termination Analysis** button of the *Termination Window* opens the Termination Analysis Window which records data computed by ✓`eriFun`'s automated termination analysis. This window is used for system development only.

**Hint 7** The *Set Termination* command can be issued alternatively by a left mouse button double click applied to the procedure in the Program Window.

***Proof***

Opens in the *Proof Window* the prooftree assigned to the selected lemma.

**Note 1** *Proof* does not apply to

⇨ predefined program items,

⇨ system generated program elements except *projection-lemmas*,

⇨ ignored lemmas.

**Hint 1** The *Proof* command can be issued alternatively by a left mouse button double click applied to a lemma in the *Program Window*, if the lemma has a non-initial prooftree.

## *Verify*

Opens in the *Proof Window* the prooftree assigned to the selected lemma or to a termination hypothesis of the selected procedure and starts a verification of this lemma or the termination hypothesis respectively.

**Note 1** *Verify* only applies
⇨ to non-verified program elements
⇨ having an initial prooftree.

**Hint 1** The *Verify* command can be issued alternatively by a left mouse button double click applied to a lemma or a procedure in the *Program Window*.

## *Verify All*

Applies the *Verify* command to all lemmas and all termination hypotheses of the actual program in the order given in the *Program Window*, until the *Verify* command fails.

**Hint 1** The system rings the bell when *Verify* fails. The *Verify All* command is particularly useful after reading an *fp*-file.

## *Copy Name*

Copies the *name* of the selected program item into the clipboard.

## *Properties*

Opens the *Program Property Window* of the selected program item.

To **view the properties of a program item**, choose the

- ⇨ **Pretty Print** tab to see a pretty print of the selected program element,
- ⇨ **Usage** tab to see the program elements which use the selected program element in the **Used by**-column and the program elements which are used by the selected program element in the **Uses**-column,
- ⇨ **Termination** tab to see the **recursion position set** and its associated **relation description** of the selected *data structure*,
- ⇨ **Termination** tab to see the **recursion position sets**, their associated **relation descriptions** and the **execution guard** of the selected *procedure*,
- ⇨ **Clauses** tab to see the selected lemma represented as a set of **clauses**,
- ⇨ **Comment** tab to see the **comment** of the selected program item.

and

choose the **Termination** tab and

- ⇨ push the **Termination Details** button to open the **Termination Window** of the selected recursively defined procedure,
- ⇨ push the **Synthesis Details** button to open the **Synthesis Details Window** of the selected **difference-procedure**.

**Note 1**

- ⇨ The **Termination Details** button is active only for *recursively defined* procedures which are already *used in proofs*. Use the *Set Termination* command of the *Program Menu* to open the *Termination Window* for *non-used* recursively defined procedures.
- ⇨ The **Synthesis Details** button is active only for difference-procedures *not belonging to a selector function* which are already *used in proofs*. Use the *Set Termination* command of the *Program Menu* to open the *Synthesis Details Window* for *non-used* difference-procedures not belonging to a selector function.

**Hint 1** *Procedures* are always pretty-printed in programming notation. *Quantifications* are always pretty-printed in functional notation.

**Hint 2** Each *recursively* defined data structure and each *recursively* defined procedure is associated with a so-called *relation description*. The relation description of a data structure defines the well-founded *structural relation* of this data structure, and the relation description of a procedure defines a *recursion relation* of this procedure, which is *well-founded* iff the procedure terminates. Relation descriptions defining well-founded relations are used to create *induction axioms*.

The *execution guard* of a procedure defines a requirement in terms of the procedure parameters which is *true* iff a procedure call does not result in a *recursive* call of the *same* procedure.

**Hint 3** In clausal notation, all negative literals are placed to the left of the `->` - arrow, and all positive literals are placed to the right of `->`. Equations are displayed as `A=B`, `A<=>B` or `A=>B` indicating the *orientation* of the equations used by ✓`eriFun`'s conditional term rewriting. Universally quantified variables are written with an apostrophe.

**Hint 4** A *Program Property Window* has the same format as the *Program Viewer*. Each *Program Property Window* exclusively belongs to a program item, whereas the *Program Viewer* displays the properties of the program item just selected in the *Program Window*.

***Copy***
Copies the definition of the selected program element into the clipboard.

# Proof

## About Proof

The *Proof Menu* provides all commands
- to edit a prooftree,
- to copy a prooftree, and
- to explore a prooftree.

Commands are selected from the menu bar or from the context menu.

Prooftrees are opened in the *Proof Window*
- using the *Proof* command from the *Program Menu*,
- using the *Verify* command from the *Program Menu*,
- using the *Verify All* command from the *Program Menu*,
- by pushing the *Proof* button in a *Termination Window*, or
- by pushing the *Proof* button in a *Synthesis Details Window*.

***Sequents***

Closed quantifications are proved in ✓eriFun using the so-called *HPL-calculus* (abbreviating *Hypotheses*, *Programs* and *Lemmas*). The formulas of the *HPL*-calculus are *sequents* of the form

$$(1)\quad \mathtt{h_1,\ldots,h_n,\ all\ldots\ ih_1,\ldots,all\ldots\ ih_l} \vdash \mathtt{goal}$$

where the expressions $\mathtt{h_i}$ (called the *hypotheses*) are literals, the expressions $\mathtt{all\ldots\ ih_k}$ (called the *induction hypotheses*) are quantifications, and the expression `goal` (called the *goalterm*) is a boolean term.

A sequent like *(1)* is *true* iff the value of $\mathtt{goal'}$, which is obtained from `goal` by substituting any values for the free variables of the sequent, is `true` whenever the values of $\mathtt{h_i'}$ and $\mathtt{ih_k'}$ are `true`, where $\mathtt{h_i'}$ and $\mathtt{ih_k'}$ are obtained from $\mathtt{h_i}$ and $\mathtt{ih_k}$ by substituting the same values for the free variables of the sequent plus substituting also *some* values for the universally quantified variables in the induction hypotheses.

The semantics of sequents respect the interpretation of the data structures and procedures and is formally defined by:

A sequent

$$\ldots\wedge\mathtt{h_i}\wedge\ldots\quad\ldots\wedge\ldots\mathtt{all\ldots\ z:sort_z}\ldots\mathtt{ih_k}\wedge\ldots\quad\vdash\mathtt{goal}$$

is *true* iff

$$\ldots\wedge \mathit{eval}(\mathtt{h_i[\ldots,x/q,\ldots]}) = \mathtt{true}\wedge\ldots$$
$$\ldots\wedge \mathit{eval}(\mathtt{ih_k[\ldots,x/q,\ldots,z/r,\ldots]}) = \mathtt{true}\wedge\ldots$$
$$\rightarrow$$
$$\mathit{eval}(\mathtt{goal[\ldots,x/q,\ldots]}) = \mathtt{true}$$

for all $\ldots,\mathtt{q},\ldots\in\mathcal{V}al$ and some $\ldots,\mathtt{r},\ldots\in\mathcal{V}al$, where variables like `x` stand for the *free* variables of the sequent.

***Prooftrees***

To prove a closed quantification an *initial prooftree* is created consisting of the root node only.

If the closed quantification is a termination hypothesis generated for a recursive call in the body of a procedure, the root node is *labeled* with a sequent

$$(2) \quad \texttt{h}_1\texttt{,...,h}_n \vdash \texttt{goal}\,,$$

where the $\texttt{h}_i$ are the literals which lead to the recursive call, and `goal` represents the ordering requirement for the parameters of the initial and the recursive call.

Otherwise, the closed quantification is of the form

$$(3) \quad \texttt{all}\ \texttt{x}_1\texttt{:sort}_1\texttt{,...,x}_n\texttt{:sort}_n\ \texttt{goal}\,,$$

representing a recursion elimination formula or the body of a lemma, and then the root node is *labeled* with a sequent

$$(4) \quad \vdash \texttt{goal}\,.$$

The descendants of a node in a prooftree, called *proofnodes* for short, are *labeled* with sequents, such that

$$(5) \quad \texttt{son}_1 \wedge \ldots \wedge \texttt{son}_n \wedge \texttt{vprog}\ (\wedge\ \texttt{lemma}) \rightarrow \texttt{father}$$

holds, where

- `father` is the label of an inner proofnode,
- $\texttt{son}_1, \ldots, \texttt{son}_n$ are the labels of its descendants,
- `vprog` denotes the set of lemmas plus the axioms derived from the data structures and the procedures of the verified program at time when the descendants of the proofnode are created, and
- `lemma` denotes an optional non-verified lemma used to create a descendant of the proofnode.

The sequents which label a proofnode are called *proofnode-sequents*. A prooftree the root of which is labeled with a sequent `S` is called a *prooftree for* `S`.

A prooftree is *closed* iff all leafs of the prooftree are labeled with a proofnode-sequent of the form

$$(6)\quad \mathtt{h_1,\ldots,h_n,\ all\ldots\ ih_1,\ \ldots\ ,\ all\ldots\ ih_l} \vdash \mathtt{true\ .}$$

A closed prooftree is *finished* iff only verified lemmas are used in the prooftree, and consequently a sequent is true, if a finished prooftree for this sequent exists.

Proofnode-sequents of the form *(1)* are represented by triples <*H*, *IH*, `goal`> in the *Proof Window*, where *H* = {$\mathtt{h_1}$ , ... , $\mathtt{h_n}$} and *IH* = {`all...` $\mathtt{ih_1}$ , ... , `all...` $\mathtt{ih_k}$}. The sets *H* and *IH* of the proofnodes are displayed in the *Proof Window* or not, depending on the user's choice. Depending on the user's choice, the *goalterms* `goal` of the proofnodes are displayed in the *Proof Window* for *all* nodes of the prooftree or for the *leafs only*. The *goalterm* `goal` of a proofnode is displayed in pretty print in the *Proof Viewer* and also in the *Proof Property Window* of that node. These windows also display the sets *H* and *IH* of the proofnode in pretty print and the set *IH* as a set of clauses.

***Developing Prooftrees***

The *proof rules* of the *HPL*-calculus are provided in the *Proof Rules Submenu* of the *Proof Menu*. By applying these proof rules to the leaves of a prooftree, this prooftree is expanded step by step. If eventually a *finished* prooftree is created, a proof for the sequent at the root node is obtained and the truth of this sequent is established.

The proof rule *Delete Hypotheses* satisfies the prooftree invariant *(5)*. All other proof rules even care for the *additional* invariant

$$(7)\quad \texttt{father} \wedge \texttt{vprog}\ (\wedge\ \texttt{lemma}) \rightarrow \texttt{son}_1 \wedge \ldots \wedge \texttt{son}_n\ .$$

A prooftree is edited in the *Proof Window* by

*(i)* selecting a leaf of the prooftree,
*(ii)* choosing a proof rule from the *Proof Rules Submenu*, and
*(iii)* providing further input, if required for the chosen proof rule.

When applying a proof rule to a leaf, the system expands the prooftree at that leaf by computing the descendant nodes for the leaf according to the chosen proof rule. A proofnode to which the proof rule *XYZ* has been applied is called an *XYZ*-node.

***Status of Proofnodes***

Proofnodes are assigned a *status* to indicate the development of a proof-tree so far. The status is represented in the *Proof Window* by a color according to the following assignments:

A proofnode has status

- *unprovable* iff the proofnode is
  ⇨ a *Delete Hypotheses*-node with a *disproved* descendant, or
  ⇨ a *Refute*-node with a *proved* descendant, or
  ⇨ a *Use Lemma*- or *Apply Equation*-node using a *falsified* lemma,
- *disproved* iff the proofnode
  ⇨ is not a *Delete Hypotheses*-node and has a *disproved* descendant, or
  ⇨ the proofnode is a leaf with an empty set of hypotheses and the goalterm `false`,
- *proved* iff the subtree rooted in the proofnode is finished,
- *completed* iff the subtree rooted in the proofnode is closed but unfinished, and no *falsified* lemmas are used in this subtree,
- *unproved* in all other cases.

Consequently, a proofnode with status

- *unprovable* is labeled with a sequent the *truth* or *falsity* of which has not been established so far,
- *disproved* is labeled with a *false* sequent,
- *proved* is labeled with a *true* sequent,
- *completed* is labeled with a sequent which is *true* if all lemmas used in the subtree rooted in the proofnode are also *true*,
- *unproved* is labeled with a sequent the *truth* or *falsity* of which has not been established so far.

*Disproved* proofnodes are created by the *Refute*-command of the *Proof Menu*. This command works like a proof rule but lacks to satisfy the prooftree invariant *(5)*. Instead, only the additional prooftree invariant *(7)* holds, i.e.

$$(8)\quad \texttt{father} \wedge \texttt{vprog} \rightarrow \texttt{son}\ .$$

From a prooftree rooted in a *completed* proofnode, a finished prooftree may be obtained without a further development of the prooftree but by verifying the non-verified lemmas used in this prooftree instead.

*Unprovable* and *unproved* proofnodes both are labeled with a sequent the *truth* or *falsity* of which has not been established so far. However, such proofnodes differ in the following way:

From a prooftree rooted in an *unproved* proofnode, a finished prooftree may be obtained by successive applications of appropriate proof rules to the leaves of the prooftree under development and verification of non-verified lemmas used in this prooftree. Alternatively, a prooftree may be obtained which refutes the sequent of the *unproved* proofnode.

Both alternatives are impossible for a prooftree containing an *unprovable* proofnode. The only way to obtain a *finished* prooftree from such a prooftree (or to obtain a prooftree which refutes the root sequent of this prooftree) is to remove the *unprovable* proofnode using the *Prune* command of the *Proof Menu*, and then to continue with another proof rule.

<>

## *Proof Rules*

Opens the *Proof Rules Submenu* of the proof editor.

## About Proof Rules

The *Proof Rules Submenu* provides a set of proof rules for editing proof-trees in the Proof Window.

Commands are selected from the menu bar or from the context menu.

A prooftree is edited by
*(i)* selecting a leaf in the prooftree,
*(ii)* choosing a proof rule, and
*(iii)* providing further input, if required for the chosen proof rule.

When applying a proof rule to a leaf, the system expands the prooftree at that leaf by computing the descendant nodes for the leaf according to the chosen proof rule.

The proof rules
- *Simplification*,
- *Weak Simplification*,
- *Normalization*,
- *Weak Normalization*,
- *Inconsistency* and
- the command *Refute* from the *Proof Menu*

are called *computed rules*. A leaf of a prooftree to which a computed rule is applied becomes a *computed proofnode*.

Each *goalterm* of a descendant of a *computed* proofnode is computed by an automated first-order theorem prover, called the *Symbolic Evaluator*. The *Symbolic Evaluator* may use hypotheses and induction hypotheses of the proofnode-sequent and may also use the lemmas and the definitions of the data structures and procedures of the verified program upon symbolic evaluation.

When active, the *Symbolic Evaluator* opens a window to display statistical data about the progress of the symbolic evaluation.

After execution of a proof rule, ✓`eriFun` executes or at least suggests a further proof rule which is computed by the *next-rule heuristic*, i.e. a proof rule which seems heuristically useful to continue with. A user may dismiss such a suggestion by pushing the *Cancel* button in the proof rule's input dialog (or by pushing *Cancel* in the window opened by the *Symbolic Evaluator*). Alternatively, the *Prune* command from the *Proof Menu* can be used for the proofnode to which „next-rule" has been applied, if he or she feels uncomfortable with the result of the suggestion.

**Simplification**
Applies the *Symbolic Evaluator* to the goalterm of the selected proof-node-sequent.

When active, the *Symbolic Evaluator* opens the *Simplification Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Simplification Window* use
⇨ the **Cancel** button to withdraw,
⇨ the **Stop** button to stop the *Symbolic Evaluator*.

**Hint 1**
⇨ The application of the *Simplification* rule is displayed as *Stopped Simplification* in the *Proof Window*, if the *Symbolic Evaluator* was stopped during operation.
⇨ The continuation of a stopped evaluation by a further application of the *Simplification* rule may yield a different result compared to the result obtained by a non-stopped application of *Simplification*.
⇨ The use of the *Stop* button is not recommended, except in the (very rare) case when the *Symbolic Evaluator* does not terminate.

**Weak Simplification**

Applies the *Symbolic Evaluator* to the goalterm of the selected proof-node-sequent without unfolding calls of *recursively* defined procedures.

When active, the *Symbolic Evaluator* opens the *Weak Simplification Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Weak Simplification Window* use

⇨ the **Cancel** button to withdraw,

⇨ the **Stop** button to stop the *Symbolic Evaluator*.

**Hint 1**

⇨ The application of the *Weak Simplification* rule is displayed as *Stopped Weak Simplification* in the *Proof Window*, if the *Symbolic Evaluator* was stopped during operation.

⇨ The continuation of a stopped evaluation by a further application of the *Weak Simplification* rule may yield a different result compared to the result obtained by a non-stopped application of *Weak Simplification*.

⇨ The use of the *Stop* button is not recommended.

**Normalization**

Applies the *Symbolic Evaluator* to the goalterm of the selected proof-node-sequent without unfolding procedure calls.

When active, the *Symbolic Evaluator* opens the *Normalization Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Normalization Window* use

⇨ the **Cancel** button to withdraw,

⇨ the **Stop** button to stop the *Symbolic Evaluator*.

**Hint 1**

⇨ The application of the *Normalization* rule is displayed as *Stopped Normalization* in the *Proof Window*, if the *Symbolic Evaluator* was stopped during operation.

⇨ The continuation of a stopped evaluation by a further application of the *Normalization* rule may yield a different result compared to the result obtained by a non-stopped application of *Normalization*.

⇨ The use of the *Stop* button is not recommended.

**Weak Normalization**

Applies the *Symbolic Evaluator* to the goalterm of the selected proof-node-sequent without unfolding procedure calls and without using lemmas and induction hypotheses.

When active, the *Symbolic Evaluator* opens the *Weak Normalization Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Weak Normalization Window* use

⇨ the **Cancel** button to withdraw,

⇨ the **Stop** button to stop the *Symbolic Evaluator*.

**Hint 1**

⇨ The application of the *Weak Normalization* rule is displayed as *Stopped Weak Normalization* in the *Proof Window*, if the *Symbolic Evaluator* was stopped during operation.

⇨ The continuation of a stopped evaluation by a further application of the *Weak Normalization* rule may yield a different result compared to the result obtained by a non-stopped application of *Weak Normalization*.

⇨ The use of the *Stop* button is not recommended.

**( Inconsistency )**
Applies the *Symbolic Evaluator* to a literal of the hypotheses set H of the selected proofnode-sequent, if this literal can be evaluated to *false*.

When active, the *Symbolic Evaluator* opens the *Inconsistency Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Inconsistency Window* use
⇨ the **Cancel** button to withdraw.

**Note 1** *Inconsistency* is only called by the next-rule-heuristic and cannot be called directly.

**Case Analysis**

Creates new nodes as descendants of the selected proofnode, which represent a case analysis wrt. a user provided case-term.

To **provide a case-term**

⇨ edit the case-term in the **Caseterm** field of the input dialog respecting the list of **Available Variables**, or

⇨ press `ctrl-P`, if the case-term has been **copied** into the clipboard before.

**Note 1** Only *if*-free non-boolean terms or boolean terms different from *true* and *false* can be used as a case-term.

**Hint 1** ✓`eriFun` performs a *propositional* case analysis for a boolean case-term and a *structural* case analysis for all other case-terms by a modification of the proofnode-sequent's goalterm.

**Use Lemma**

Applies a user provided instance of a user chosen lemma or induction hypothesis to the goalterm of the selected proofnode-sequent.

To **use a lemma or an induction hypothesis** in the input dialog
- ⇨ choose the lemma or induction hypothesis from the list of **Available Lemmas**,
- ⇨ from the list of **Available Positions** select a position where the lemma or induction hypothesis is to be applied,
- ⇨ provide a **Substitution** for the variables of the selected lemma or induction hypothesis respecting the list of **Available Variables**,
- ⇨ push the **Cancel** button to withdraw,
- ⇨ push the **OK** button to quit.

To **provide a Substitution** for the variables of the selected lemma or induction hypothesis
- ⇨ edit the substitution in the input field of the dialog, or
- ⇨ click any position in the input field of the dialog and press `ctrl-P`, if the substitution has been **copied** into the clipboard before, or
- ⇨ click the area which lists the variables to be substituted and choose *Paste* from the context menu, if the substitution has been **copied** into the clipboard before.

**Hint 1** Ignored and falsified lemmas are not available in *Use Lemma*.

## Unfold Procedure

Replaces a user chosen procedure call in the goalterm of the selected proofnode-sequent by the instance of the procedure body which is obtained by substituting the formal parameters in the body with the actual parameters of the procedure call.

To **unfold a procedure call**

⇨ select the **procedure name** and open the list associated with this name in the upper half of the input dialog,

⇨ select the **position** of the procedure call in the goalterm, which is displayed in the lower half of the input dialog,

⇨ push the **Cancel** button to withdraw,

⇨ push the **OK** button to quit.

**Hint 1** Positions of procedure calls of symmetric or commutative function symbols appear twice in the list, one displayed normal and the other one marked with an asterisk, where a selection of the latter position results in an unfold of the procedure call with *arguments interchanged*.

**Apply Equation**

Performs an equality replacement step in the goalterm of the selected proofnode-sequent using a user provided instance of a user specified equation.

To **apply an equation** from an *Induction Hypothesis* or a *Lemma*

- ⇨ open one of the lists associated with the labels **Induction Hypotheses** or **Lemmas** in the upper half of the input dialog by clicking the ⊞-box or by a left-mouse double-click,
- ⇨ open one of the *Induction Hypotheses* or *Lemmas* in the opened list which contains the equation to be applied,
- ⇨ open one of the *clauses* belonging to the opened Induction Hypothesis or Lemma which contains the equation to be applied,
- ⇨ open one of the *equations* in the opened clause which shall be applied,
- ⇨ open or select one of the *positions* in the opened equation to locate the place where the equation should be applied in the goalterm (which is displayed in the lower half of the input dialog),
- ⇨ select one *substitution* (if any) in the opened position to uniquely bind the variables of the selected equation, if this equation or the subterm to be replaced contains **commutative function symbols**,
- ⇨ provide a **Substitution** for the unbound variables of the selected equation respecting the list of **Available Variables**,
- ⇨ push the **Cancel** button to withdraw,
- ⇨ push the **OK** button to quit.

To **provide a Substitution** for the unbound variables of the selected equation

- ⇨ edit the substitution in the input field of the dialog, or
- ⇨ click any position in the input field of the dialog and press `ctrl-P`, if the substitution has been **copied** into the clipboard before, or
- ⇨ click the area which lists the variables to be substituted and choose *Paste* from the context menu, if the substitution has been **copied** into the clipboard before.

To **apply an equation** from the set of **hypotheses** or from the **local hypotheses** of the *goalterm*

⇨ open one of the lists associated with the labels **Global Hypotheses** or **Local Hypotheses** in the upper half of the input dialog by clicking the ⊞-box or by a left-mouse double-click,

⇨ open one of the *equations* in the opened list which shall be applied,

⇨ select one of the *positions* in the opened equation to locate the place where the equation should be applied in the goalterm (which is displayed in the lower half of the input dialog),

⇨ push the **Cancel** button to withdraw,

⇨ push the **OK** button to quit.

**Hint 1** Ignored and falsified lemmas are not available in *Apply Equation*.

## Induction

Creates new proofnodes as descendants of the selected proofnode which represent the base and step case sequents of a user chosen induction axiom.

To **choose an induction axiom**

⇨ open one of the lists associated with the name of a *data structure* or a recursively defined *procedure* in the upper part of the input dialog,

⇨ open one of the lists associated with a **Recursion Position Set** of the selected program element,

⇨ select one **substitution** in the opened list associating the variables of the induction axiom with the free variables of the proofnode-sequent,

⇨ push the **Edit Theta** button if required to provide a substitution for the unbound variables of the selected induction axiom respecting the list of **Available Variables**,

⇨ push the **Cancel** button to withdraw,

⇨ push the **OK** button to quit.

**Note 1** *Induction* does not apply to

⇨ proofnodes with a non-empty set *H* of hypotheses,

⇨ proofnodes with a non-empty set of *IH* induction hypotheses.

**Insert Hypotheses**

Creates new nodes as descendants of the selected proofnode, which represent a case analysis wrt. a user provided list of hypotheses.

To **provide a list of hypotheses**

⇨ edit the list of hypotheses (separated by commas) in the **Hypotheses** field of the input dialog respecting the list of **Available Variables**, or

⇨ press `ctrl-P`, if the list of hypotheses has been **copied** into the clipboard before.

**Note 1** Only *if*-free non-boolean terms or boolean terms different from *true* and *false* can be used in the list of hypotheses.

**Hint 1** ✓`eriFun` performs a *propositional* case analysis for a boolean hypothesis and a *structural* case analysis for all other hypotheses by a modification of the proofnode-sequent's set of hypotheses.

**Hint 2** *Insert Hypothesis* is particularly useful to support the execution of *several* calls of procedures (sharing the same recursion structure) in a subsequent use of the *Simplification* rule. However, the use of *Insert Hypothesis* bears the risk of *over-evaluation*. Use *Case Analysis* and/or *Unfold Procedure* to avoid over-evaluations.

**Move Hypothesis**

Moves a user specified literal from the hypotheses set H of the selected proofnode-sequent to the goalterm of that node.

To **move a hypothesis**

⇨ select one hypothesis in the input dialog,
⇨ push the **Cancel** button to withdraw,
⇨ push the **OK** button to quit.

**Hint 1** *Move Hypothesis* is particularly useful, if nested inductions are required in a proof: As the *Induction* rule does not apply if the set *H* of hypotheses is not empty, *Move Hypothesis* can be used to obtain an empty set of hypotheses while preserving the equivalence between the *Move Hypothesis*-node and its descendant.

**Delete Hypotheses**
Either deletes *some* user chosen members in the hypotheses set H or *all* members in the hypotheses set H and in the induction hypotheses set IH of the selected proofnode-sequent.

To **delete (induction) hypotheses** from a proofnode-sequent
⇨ select one or several hypotheses from *H* in the input dialog using the **ctrl**- or the **shift**-key for a joint selection,
⇨ push the **Cancel** button to withdraw,
⇨ push the **All** button to delete all hypotheses from *H* and *IH*,
⇨ push the **OK** button to delete only the hypotheses selected in the input dialog.

**Hint 1** *Delete Hypotheses* is used to get rid of irrelevant conditions in a termination proof which the system regards as relevant.

*Delete Hypotheses* is useful, if nested inductions are required in a proof, because the *Induction* rule does not apply if the sets *H* of hypotheses or *IH* of induction hypotheses are not empty. *Delete Hypotheses* implements a *generalization* of the proofnode-sequent. Use *Move Hypothesis* and *Use Lemma* (applied to induction hypotheses) to avoid *over-generalizations*, i.e. the creation of *false* proofnode-sequents.

***(Show Marks)***
Displays data annotating the goalterm of the selected proofnode to control the *Symbolic Evaluator*.

**Note 1** *Show Marks* is available for system development only.

## *Show / Hide Details*

Shows/hides in the *Proof Window* the set H of hypotheses and the set IH of induction hypotheses of the selected proofnode-sequent. Also uncovers/covers additional details about the proof rule which has been applied to the selected proofnode.

To **show/hide the additional details associated with a proof rule application**, in the list which has been opened with the *Show Details* command

⇨ click the **name** of the proof rule with the center mouse button, or

⇨ click the ⊞-box/⊟-box left to the **name** of the proof rule with the left mouse button.

**Hint 1** If the selected node is a computed proofnode, the additional details consist of

⇨ the set **Used IH** of induction hypotheses used in the symbolic evaluation associated with the proofnode,

⇨ the set **Used Lemmas** of the names of the lemmas used in the symbolic evaluation associated with the proofnode,

⇨ statistical data (except memory data) which has been assembled when the symbolic evaluation associated with the proofnode was created.

If the selected node is not a computed proofnode, the additional details consist of

⇨ the system computed or user provided input of the proof rule application which has created the descendant of the selected proofnode.

To **copy system computed or user provided input of a proof rule application into the clipboard**

⇨ select the item to be copied from the list opened by *Show Details*,

⇨ choose *Copy* from the *Proof Menu*.

**Hint 2** For computed proofnodes and for leafs of a prooftree, the *Show/Hide Details* command also provides the actual setting of the parameters which control symbolic evaluation. This data is used for system development only.

**Hint 3** The *Show/Hide Details* command is alternatively issued by a center mouse button click applied to the proof node.

## *Hide / Show Rules*

Assigns in the *Proof Window* each node of a prooftree the goalterm of that node. / Assigns in the *Proof Window* each inner node of a prooftree the name of the proof rule, which has been applied to the node, plus additional details of this rule application.

**Hint 1** If the selected node is a computed proofnode, the additional details consist of

⇨ the set of induction hypotheses and the set of the names of the lemmas used in the symbolic evaluation associated with the proofnode

If the selected node is not a computed proofnode, the additional details consist of

⇨ some of the system computed or user provided input of the proof rule application which has created the descendant of the selected proofnode.

**Hint 2** Except for computed rules applied to the *root* of a prooftree, rule names appear with an asterisk if the rule application was issued by the next-rule heuristic or by the *Verify* command from the *Program Menu*.

### *(Induction Heuristic)*

Opens the *Induction Heuristic Window* to display a list of relation descriptions representing induction axioms, which seem heuristically useful when applying the *Induction* rule to the selected proofnode.

**Note 1** This command is available for system development only. *Induction Heuristic* does not apply to

⇨ proofnodes with a non-empty set *H* of hypotheses,

⇨ proofnodes with a non-empty set *IH* of induction hypotheses.

**Hint 1** *Induction Heuristic* is used to monitor the results of ✓eriFun's induction heuristic. The *Verify* and the *Verify All* commands of the *Program Menu* use the first relation description in the list computed by *Induction Heuristic*.

## *Refute*

Applies the *Symbolic Evaluator* to a user provided instance of the selected proofnode-sequent.

To **refute a proofnode-sequent**
⇨ in the input dialog provide a **Substitution** for the free variables of the proofnode-sequent respecting the list of **Available Variables**,
⇨ push the **Cancel** button to withdraw,
⇨ push the **OK** button to quit.

To **provide a Substitution** for the free variables of the proofnode-sequent
⇨ edit the substitution in the input field of the dialog, or
⇨ click any position in the input field of the dialog and press `ctrl-P`, if the substitution has been **copied** into the clipboard before, or
⇨ click the area which lists the variables to be substituted and choose *Paste* from the context menu, if the substitution has been **copied** into the clipboard before.

When active, the *Symbolic Evaluator* opens the *Refute Window* to display **statistical data** about the progress of the symbolic evaluation. **While the *Symbolic Evaluator* is operating**, in the *Refute Window* use
⇨ the **Cancel** button to withdraw,
⇨ the **Stop** button to stop the *Symbolic Evaluator*.

**Hint 1**

⇨ The application of the *Refute* command is displayed as *Stopped Refutation* in the *Proof Window*, if the *Symbolic Evaluator* was stopped during operation.

⇨ The continuation of a stopped evaluation by a further application of the *Refute* command may yield a different result compared to the result obtained by a non-stopped application of *Refute*.

⇨ The use of the *Stop* button is not recommended, except in the (very rare) case when the *Symbolic Evaluator* does not terminate.

**Hint 2** If successful, *Refute* proves the *negation* of the proofnode-sequent. *Refute* is particularly useful for debugging the actual program.

***(Set Control)***

Opens a dialog to alter

⇨ the parameters which control the *Symbolic Evaluator*,

⇨ the flag to enable/disable the *lemma-filter-heuristic*,

⇨ the flag to enable/disable the *assumption-heuristic*.

The new settings are only effective for the execution of the first computed rule following *Set Control*.

**Note 1** *Set Control* is available for system development only.

## *Create Lemma*

Inserts a lemma into the actual program with a user specified name and the body obtained from the goalterm of the selected proofnode-sequent. The created lemma is assigned a copy of the prooftree rooted in the selected proofnode.

**Note 1** *Create Lemma* does not apply to

⇨ proofnodes with a non-empty set *H* of hypotheses,

⇨ proofnodes with a non-empty set *IH* of induction hypotheses.

**Hint 1** *Create Lemma* is particularly useful

⇨ if *alternative developments* of a proof shall be explored without spoiling the prooftree created so far,

⇨ to make the sequent of an induction subproof in a *nested induction* available for symbolic evaluation,

⇨ to make a verified *termination hypothesis* available for symbolic evaluation.

## *Prune*

Prunes the prooftree at the selected proofnode.

**Note 1** *Prune* does not apply
⇨ to a leaf of a prooftree,
⇨ if the program element to which the prooftree belongs is verified and is *used* in other proofs.

**Hint 1** All program elements, which use a program element, are listed in the **Used by**-column of the **Usage** tab in the Program Viewer or the program element's Program Property Window.

**Hint 2** Since ✓eriFun does not provide an undo-feature, the system asks for confirmation before pruning the prooftree.

## *Evaluation*

Opens in the *Evaluation Window* the symbolic evaluation which has been computed when the selected *computed* proofnode was created and displays the first term of the symbolic evaluation in the *Evaluation Viewer*.

To **browse through a symbolic evaluation**

⇨ if the inspection should *not* start from the beginning, select the term in the **Evaluation Window** from which the inspection of the symbolic evaluation should start,

⇨ open the **Evaluation Viewer**,

⇨ select the **Goal** tab to see a pretty print of the term in the symbolic evaluation which is modified by the next applicable evaluation rule,

⇨ recognize the **headline** of the *Evaluation Viewer* to see the name of the **evaluation rule** to be applied next to the highlighted subterm,

⇨ use the ↓ - key to proceed to the term resulting from the application of the next applicable evaluation rule,

⇨ use the ↑ - key to step back to the term from which the displayed term resulted by the most recent evaluation step.

**Hint 1** In the *Evaluation Window*, a symbolic evaluation is preceded by a list consisting of

⇨ the **name** of the computed rule which has created the symbolic evaluation,

⇨ a *multiset* **A** of the induction hypotheses and the names of the verified lemmas used in the symbolic evaluation,

⇨ statistical data (except memory data) which has been assembled when the symbolic evaluation was created.

**Hint 2** Opening a symbolic evaluation in the *Evaluation Window* may be delayed proportional to the time required for computing the symbolic evaluation.

## *Properties*

Opens the *Proof Property Window* of the selected proofnode.

To **view the properties of a proofnode**

- ⇨ use the **Goal** tab to see a pretty print of the selected proofnode's **goalterm**,
- ⇨ use the **Hypotheses** tab to see the selected proofnode's set H of **hypotheses**, where hypotheses *used* in the prooftree rooted in the selected proofnode are displayed *red*,
- ⇨ use the **Induction Hypotheses** tab to see a pretty print of the selected proofnode's set IH of **induction hypotheses**,
- ⇨ use the **Clauses** tab to see the set of **clauses** computed for the selected proofnode's set IH of **induction hypotheses**.
- ⇨ use the **Available Lemmas** tab to see the list of **verified** lemmas which are available for symbolic evaluation of the selected proofnode's goalterm according to ✓eriFun's **lemma-filter-heuristic**.

**Hint 1** Universally quantified variables of the induction hypotheses are written with asterisks. In clausal notation, all negative literals are placed to the left of the `->` - arrow, and all positive literals are placed to the right of `->`. Equations are displayed as `A=B`, `A<=>B` or `A=>B` indicating the *orientation* of the equations used by ✓eriFun's conditional term rewriting.

The list of lemmas displayed when using the *Available Lemmas* tab depends on the set of *verified* lemmas which are available at *viewing time*.

**Hint 2** A *Proof Property Window* has the same format as the *Proof Viewer*. Each *Proof Property Window* exclusively belongs to a proofnode, whereas the *Proof Viewer* displays the properties of the proofnode just selected in the *Proof Window*.

***Copy***
Copies the goalterm of the selected proofnode into the clipboard.

# Evaluation

## About Evaluation

The *Evaluation Menu* provides all commands to

- inspect a symbolic evaluation in detail.

Commands are selected from the menu bar or from the context menu.

Symbolic evaluations are opened in the *Evaluation Window* using the *Evaluation* command from the *Proof Menu*.

***The Symbolic Evaluator***
is the automated theorem prover of ✓eriFun designed to perform all first-order inferences when verifying lemmas, termination hypotheses and recursion elimination formulas. The *Symbolic Evaluator* is also used for several other tasks in the system. The *Symbolic Evaluator* is based on the so-called *Evaluation Calculus*, i.e. a calculus consisting of a set of inference rules, called the *evaluation rules*. The formulas of the evaluation calculus are *annotated first-order terms*, i.e. first-order terms which are supplied with *control information* to guide the *Symbolic Evaluator*.

Each evaluation rule is of the form

$$(1)\quad \frac{\texttt{a_term}}{\texttt{a_term}'}\ \text{, if } \textsc{side_condition}(\texttt{a_term})\text{ ,}$$

where SIDE_CONDITION(a_term) expresses some property of the annotated term a_term. Starting with an (annotated) goalterm of some sequent, the *Symbolic Evaluator* applies the first applicable evaluation rule to this goalterm, then applies the first applicable evaluation rule to the result of the recent evaluation step and so on, until eventually an annotated term is obtained to which no further evaluation rule can be applied. The list of terms obtained thereby defines a *deduction* in the evaluation calculus, called a *symbolic evaluation* of the goalterm. To each computed proofnode of a prooftree, a symbolic evaluation of the proofnode's goalterm is assigned.

When evaluating an (annotated) goalterm of a sequent symbolically, the *Symbolic Evaluator* may use the *hypotheses* and the *induction hypotheses* of the sequent and may also use the *lemmas* and the *definitions* of the data structures and procedures of the verified program. However, to cut down the search space only those lemmas of the verified program are considered upon symbolic evaluation which are selected by ✓eriFun's *lemma-filter-heuristic* and by the *assumption-heuristic*, i.e. heuristics which exclude certain lemmas from the symbolic evaluation if they do not *seem* to contribute to a proof.

The *Symbolic Evaluator* is *sound* in the sense that each term is semantically equal to its evaluated counterpart. More precisely, all evaluation rules of the form *(1)* which are used to compute a symbolic evaluation of some computed proofnode with a proofnode-sequent of the form

$$\textit{(2)}\quad \texttt{h}_1,\ldots,\texttt{h}_n,\ \texttt{all}\ldots\ \texttt{ih}_1,\ldots,\texttt{all}\ldots\ \texttt{ih}_1\ \vdash\ \ldots$$

satisfy

$$\texttt{vprog}$$

$$\rightarrow$$

$$\texttt{h}_1,\ldots,\texttt{h}_n,\ \texttt{all}\ldots\ \texttt{ih}_1,\ldots,\texttt{all}\ldots\ \texttt{ih}_1 \vdash \textsc{plain}(\texttt{a_term}) = \textsc{plain}(\texttt{a_term}')$$

where `vprog` denotes the set of lemmas plus the axioms derived from the data structures and the procedures of the verified program at time when the symbolic evaluation is computed, and PLAIN(`a_term`) is obtained from `a_term` by stripping off all control information.

Symbolic evaluation does not necessarily terminate. This means, that the *Symbolic Evaluator* may run out of memory for some proofnode-sequents without computing a result. However, theses cases are very unlikely to happen, but if so, the user must step in to *stop* or to *cancel* the symbolic evaluation of the goalterm.

***Calling the Symbolic Evaluator***

The *Symbolic Evaluator* is a completely automated tool on which the ✓eriFun user has no direct influence, except to *stop* or to *cancel* a symbolic evaluation. However, a user may *indirectly* influence the computation of a symbolic evaluation by providing or not providing *verified* lemmas for the actual program, thus modifying the verified program.

For *system development*, the parameters which control the computation of a symbolic evaluation can be *globally* or *locally* modified using the *Customize* command from the *File Menu* or the *Set Control* command from the *Proof Menu* to tune the *Symbolic Evaluator* at the developers wish. The actual setting of the control parameters used to compute a symbolic evaluation can be viewed using the *Show Details* command from the *Proof Menu* applied to the computed proofnode or alternatively in the prologue of the symbolic evaluation opened in the *Evaluation Window*.

The *Symbolic Evaluator* is called via the computed rules of the *Proof Rules Submenu*, viz.

- *Simplification*,
- *Weak Simplification*,
- *Normalization*,
- *Weak Normalization*,
- *Inconsistency* and
- the command *Refute* from the *Proof Menu*

each of which invoke an incarnation of the *Symbolic Evaluator* parameterized for the user's need.

When active, the *Symbolic Evaluator* opens a window to display statistical data about the progress of the symbolic evaluation:

⇨ *Generated Main Nodes* (*GMN*) counts the number of evaluation steps *without* counting the steps required to compute the goalterm's subterms to be modified by the symbolic evaluation, thus giving the *length* of the symbolic evaluation,

⇨ *Successful Rule Applications* (*SRA*) counts the number of evaluation steps *including* the steps required to compute the goalterm's subterms to be modified by the symbolic evaluation,

⇨ *Average Depth* is the ratio *SRA/GMN*,

⇨ *SRA/sec* is the ratio *SRA/Evaluation Time*, thus giving the „speed“ of the *Symbolic Evaluator*,

⇨ *Evaluation Time* counts the time elapsed so far,

⇨ *Free Memory* gives the memory presently available for the system,

⇨ *Total Memory* gives the memory assigned to the system.

The statistical data (except memory data) assembled for a symbolic evaluation of a computed proofnode can be viewed using the *Show Details* command from the *Proof Menu* or alternatively in the prologue of the symbolic evaluation opened in the *Evaluation Window*.

***Exploring a Symbolic Evaluation***

In general, a ✓eriFun user does not need to recognize the deductions computed by the *Symbolic Evaluator*. However, the system makes symbolic evaluations available for exploration because users like

- to gain insight, why a certain proof idea does not work,
- to learn how the system came up with a surprising solution,
- to feel more comfortable when the system's findings are subject to inspection.

<>◇<>

### *(Show Marks)*

Displays data which annotates the terms of a symbolic evaluation to control the *Symbolic Evaluator*.

**Note 1** *Show Marks* is available for system development only.

### *(Show / Hide Details)*

Uncovers/covers data computed by the *Symbolic Evaluator* in a user selected (sub)term of a symbolic evaluation.

**Note 1** *Show/Hide Details* is available for system development only.

## *Properties*

Opens the *Evaluation Property Window* of a user selected term in a symbolic evaluation.

To **view the properties of a term in a symbolic evaluation**

- ⇨ use the **Goal** tab to see a pretty print of the selected term,
- ⇨ use the **Hypotheses** tab to see the set of **hypotheses** plus the **local hypotheses**, which are available for the selected term,
- ⇨ use the **Induction Hypotheses** tab to see a pretty print of the set of **induction hypotheses** of the proofnode-sequent to which the symbolic evaluation belongs,
- ⇨ use the **Clauses** tab to see the actual list of **clauses** and select a *clause* in the list to see the **induction hypothesis** or the *name of the* **lemma** from which the selected clause stems.

**Hint 1** The list of literals displayed when the **Hypotheses** tab is chosen is composed of the set $H$ of hypotheses of the proofnode-sequent, to which the symbolic evaluation belongs, plus the list of all *local* hypotheses which are available when the next applicable evaluation rule is executed.

*Local hypotheses* are indicated by `[l]` in the list and are given by those *if*-conditions in the selected term, which lead to the subterm to be replaced by the next applicable evaluation rule.

**Hint 2** Universally quantified variables of the induction hypotheses are written with asterisks. In clausal notation, all negative literals are placed to the left of the `->` - arrow, and all positive literals are placed to the right of `->`. Equations are displayed as `A=B`, `A<=>B` or `A=>B` indicating the *orientation* of the equations used by ✓`eriFun`'s conditional term rewriting.

The list of clauses displayed is computed from set *IH* of induction hypotheses of the proofnode-sequent, to which the symbolic evaluation belongs, plus the set of lemmas of the verified program *at time when opening* the *Evaluation Property Window*. All clauses in this list, which have been *used* in the symbolic evaluation, are displayed *normal*. All other clauses are displayed *light*. Universally quantified variables in clauses belonging to induction hypotheses are written with asterisks and universally quantified variables belonging to clauses of lemmas are written with an apostrophe.

**Hint 3** An *Evaluation Property Window* has the same format as the *Evaluation Viewer*. Each *Evaluation Property Window* exclusively belongs to a term of a symbolic evaluation, whereas the *Evaluation Viewer* displays the properties of the term just selected in the *Evaluation Window*.

***Copy***
Copies the user selected term of a symbolic evaluation into the clipboard.

# Window

## About Window

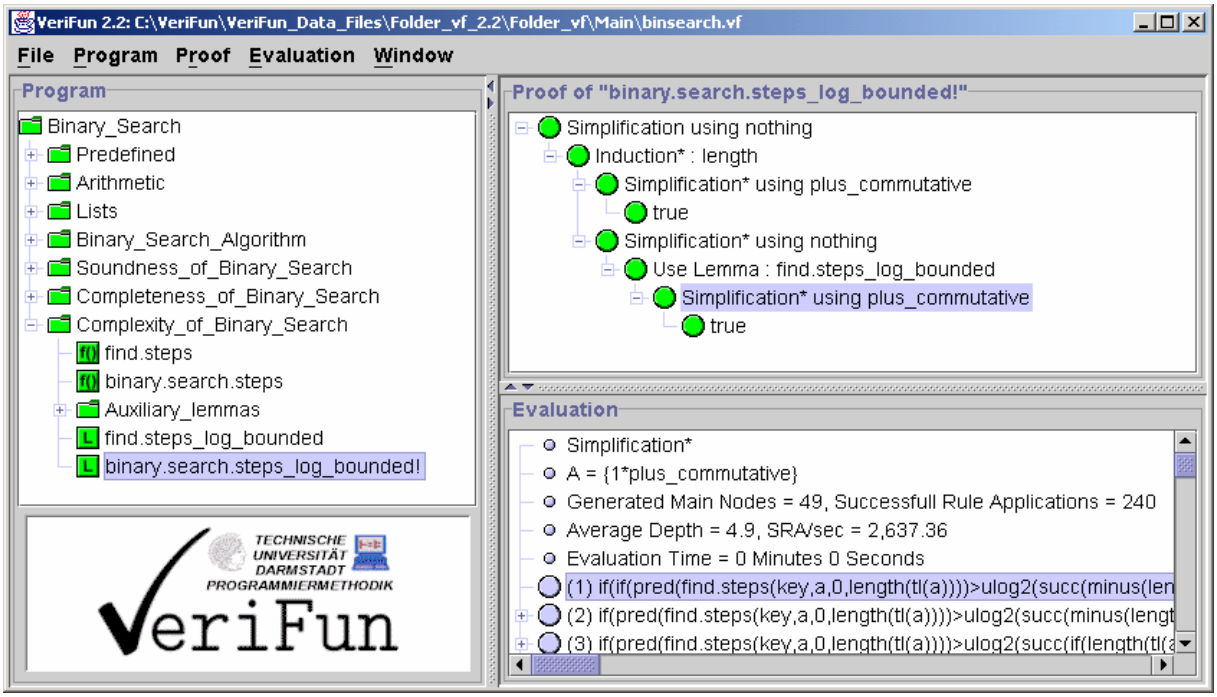
The *Window Menu* provides commands

- to arrange the main system window and the *Viewers*,
- to open/close the *Program Viewer*,
- to open/close the *Proof Viewer*,
- to open/close the *Evaluation Viewer*,
- to open/close the *Legend Viewer*,
- to close all *Property Windows*.

Commands are selected from the menu bar or from the context menu.

✓eriFun's main system window is separated into *3* subwindows, called

- the *Program Window*, which displays the actual program,
- the *Proof Window*, which displays prooftrees, and
- the *Evaluation Window*, which displays symbolic evaluations.



Each of these subwindows are assigned further windows which care for more detailed information.

***The Program Window***
is supported by the following windows:

- The *Program Viewer* displays relevant properties of the program item which is selected in the *Program Window*. See the *Properties* command of the *Program Menu* for a description of the data displayed by the *Program Viewer*.
- The *Program Property Window* displays relevant properties of a program item. These windows have the same format as the *Program Viewer* but are exclusively assigned to a certain program item. A *Program Property Window* is opened by the *Properties* command of the *Program Menu*.
- The *Termination Window* is used to submit and display data relevant for termination proofs. Prooftrees of termination hypotheses are opened in the *Proof Window* via the *Termination Window*. This window is opened (for input and for editing prooftrees of termination hypotheses) by the *Set Termination* command of the *Program Menu* or (for viewing only) via the *Termination Details* button in the *Program Viewer* or in a *Program Property Window*.
- (The *Termination Analysis Window* records data computed by ✓`eri-Fun`'s automated termination analysis. This window is opened via the *Termination Window*. The *Termination Analysis Window* is used for system development only.)
- The *Synthesis Details Window* provides data about the difference-procedures synthesized by the system. Prooftrees of recursion elimination formulas are opened in the *Proof Window* via the *Synthesis Details Window*. This window is opened (for editing prooftrees of recursion elimination formulas) by the *Set Termination* command of the *Program Menu* or (for viewing only) via the *Synthesis Details* button in the *Program Viewer* or in a *Program Property Window*.

***The Proof Window***

is supported by the following windows:

- The *Proof Viewer* displays relevant properties of the proofnode which is selected in the *Proof Window*. See the *Properties* command of the *Proof Menu* for a description of the data displayed by the *Proof Viewer*.
- The *Proof Property Window* displays relevant properties of a proofnode. These windows have the same format as the *Proof Viewer* but are exclusively assigned to a certain proofnode. A *Proof Property Window* is opened by the *Properties* command of the *Proof Menu*.

***The Evaluation Window***

is supported by the following windows:

- The *Evaluation Viewer* displays relevant properties of the term (of a symbolic evaluation) which is selected in the *Evaluation Window*. See the *Properties* command of the *Evaluation Menu* for a description of the data displayed by the *Evaluation Viewer*.
- The *Evaluation Property Window* displays relevant properties of a term (of a symbolic evaluation). These windows have the same format as the *Evaluation Viewer* but are exclusively assigned to a certain term of a symbolic evaluation. An *Evaluation Property Window* is opened by the *Properties* command of the *Evaluation Menu*.

<>

### *Tile Vertically*

Arranges the main system window, the *Program Viewer* and the *Proof Viewer* vertically on the desktop.

**Hint 1** Some JAVA installations do not display the windows correctly. In such a case, a slight alteration of the window shape will yield the proper format.

### *Tile Horizontally*

Arranges the main system window, the *Program Viewer* and the *Proof Viewer* horizontally on the desktop.

**Hint 1** Some JAVA installations do not display the windows correctly. In such a case, a slight alteration of the window shape will yield the proper format.

### *Open/Close Program Viewer*

Opens/closes the *Program Viewer*.

### *Open/Close Proof Viewer*

Opens/closes the *Proof Viewer*.

### *Open/Close Evaluation Viewer*

Opens/closes the *Evaluation Viewer*.

### *Open/Close Legend Viewer*

Opens/closes the *Legend Viewer*.

**Hint 1** The *Legend Viewer* shows the colors which are used in the *Program Window* to indicate the status of a program element and the colors used to indicate the status of a proofnode in the *Proof Window*.

### *Close Property Windows*

Closes each *Program Property Window*, each *Proof Property Window* and each *Evaluation Property Window*.

# Miscellaneous

*(1)* Most dialogs allow the usual keys `ctrl-A`, `ctrl-C`, `crtl-V` and `ctrl-X` for *select all*, *copy*, *paste* and *cut*.

*(2)* Some dialogs appear different under *Unix/Linux* vs. *Windows* platforms.

*(3)* Parsing errors may result from special characters which do not appear in print. In such a case, copy the input and use an ASCII-Editor, which makes such characters visible, for deletion.

>< >< ><